

国际信息工程先进技术译丛

WILEY

云计算技术与商业趋势

Cloud Computing: Business Trends and Technologies

[美] 伊戈尔·费恩伯格 (Igor Faynberg)
卢慧兰 (Hui-Lan Lu)
多尔·斯库勒 (Dor Skuler) 著

郎为民 王大鹏 陈 红 姚晋芳 等译







机械工业出版社
CHINA MACHINE PRESS

国际视野　科技前沿

国际信息工程先进技术译丛

《云计算技术与商业趋势》
《数字信号处理与滤波器设计》
《信号处理系统的FPGA实现》
《多处理器片上系统的硬件设计与工具集成》
《植入式电子医疗器械》
《云计算体系架构中的智能SOA平台》
《纳米CMOS集成电路中的小延迟缺陷检测》
《绿色通信与网络》
《自主式传感器系统的能量收集——设计、分析以及实践应用》
《基于视觉的自主机器人导航》
《无线神经接口的超低功耗集成电路设计》
《基于片上去耦电容的配电网络》（原书第2版）
《智能摄像机》
《车载系统和安全的数字信号处理》
《嵌入式系统设计 —— 嵌入式信息物理系统基础》（原书第2版）
《纳米封装 —— 纳米技术与电子封装》
《内容分发网络》
《全面的功能验证：完整的工业流程》
《无线**Mesh**网络架构与协议》
《**UMTS**蜂窝系统的**QoS**与**QoE**管理》
《半导体制造与过程控制基础》
《**WCDWA**原理与开发设计》
《下一代移动系统：**3G/B3G**》
《**IMS:IP**多媒体概念和服务》（原书第2版）
《下一代无线系统与网络》
《深入浅出**UMTS**无线网络建模、规划与自动优化：理论与实践》
《**HSDPA/HSUPA**技术与系统设计——第三代移动
《通信系统宽带无线接入》
《无线传感器及元器件：网络、设计与应用》
《印制电路板——设计、制造、装配与测试》
《**IPTV**与网络视频：拓展广播电视的应用范围》
《多电压**CMOS**电路设计》
《微电子技术原理、设计与应用》
《蜂窝网络高级规划与优化**2G/2.5G/3G/…**向**4G**的演进》
《基于蜂窝系统的**IMS**——融合电信领域的**VoIP**演进》
《无线网络中的合作原理与应用》
《电生理学方法与仪器入门》
《移动电视：**DVB-H**、**DMB**、**3G**系统和富媒体应用》
《环境网络：支持下一代无线业务的多域协同网络》
《基于射频工程的**UMTS**空中接口设计与网络运行》
《未来**UMTS**的体系结构与业务平台：全**IP**的**3G CDMA**网络》



WILEY



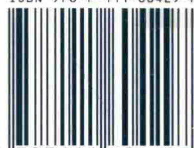
机械工业出版社微信公众号



E视界
传播电类内容　提升专业知识



科技电眼
关注电类行业动向 聚焦前沿科技



上架指导　工业技术／信息技术

ISBN 978-7-111-60429-7

定价：89.00元

国际信息工程先进技术译丛

云计算技术与商业趋势

Cloud Computing：Business Trends and Technologies

伊戈尔·费恩伯格（Igor Faynberg）
［美］　卢慧兰（Hui－Lan Lu）　著
多尔·斯库勒（Dor Skuler）
郎为民　王大鹏　陈　红　姚晋芳　等译



机　械　工　业　出　版　社

本书紧紧围绕云计算领域发展过程中的热点问题，以云计算技术与商业应用为核心，全面系统地介绍了云计算的基本原理和应用实践的最新成果。全书共分为7章，涉及云计算业务，CPU虚拟化，数据网络，网络设备，现代数据中心的云存储与结构，云内部的运营、管理与业务流程编排等内容。此外，附录A还对IETF业务与管理标准、TOSCA业务流程、REST架构风格、身份与访问管理机制进行了详细介绍。本书材料权威丰富，体系结构完整，内容新颖翔实，知识系统全面，行文通俗易懂，兼备知识性、系统性、可读性、实用性和指导性。本书可作为从事云计算、数据中心研究的电信运营商、网络运营商、应用开发人员、技术经理和电信管理人员的技术参考书或培训教材，也可作为高等院校通信与信息系统、计算机等相关专业高年级本科生或研究生教材。

图书在版编目（CIP）数据

云计算技术与商业趋势/(美）伊戈尔·费恩伯格（Igor Faynberg）等著；郎为民等译. —北京：机械工业出版社，2018.10

（国际信息工程先进技术译丛）

书名原文：Cloud Computing：Business Trends and Technologies

ISBN 978-7-111-60429-7

Ⅰ.①云…　Ⅱ.①伊…②郎…　Ⅲ.①云计算 – 商业模式 – 研究
Ⅳ.①F713.361

中国版本图书馆CIP数据核字（2018）第156143号

机械工业出版社（北京市百万庄大街22号　邮政编码100037）
策划编辑：张俊红　责任编辑：朱　林
责任校对：陈　越　封面设计：马精明
责任印制：张　博
北京华创印务有限公司印刷
2018年9月第1版第1次印刷
184mm×260mm · 15.25印张 · 484千字
标准书号：ISBN 978-7-111-60429-7
定价：89.00　元

凡购本书，如有缺页、倒页、脱页，由本社发行部调换

电话服务
服务咨询热线：010 – 88361066
读者购书热线：010 – 68326294
　　　　　　　010 – 88379203

网络服务
机 工 官 网：www.cmpbook.com
机 工 官 博：weibo.com/cmp1952
金　书　网：www.golden – book.com
教育服务网：www.cmpedu.com

封面无防伪标均为盗版

译者序

从2011年开始，我国云计算市场进入了应用全面落地阶段。从中央到地方的各级政府部门均大力支持云计算产业发展，将云计算纳入“十二五”规划重点技术。在北京、上海、深圳、杭州、无锡5个试点城市的带动下，国内涌现出了一大批的云计算基地，很多高科技园区也将云计算列为重点项目。这些云计算基地的创建为传统ICT巨头提供了更广阔的发展空间，也为更多创业型企业提供了发展机遇，促进了产业快速发展。银行、证券、政府、电力、石化等大型行业企业对安全性要求较高，比较看重云计算的大规模数据处理、海量数据安全存储等性能及可用性方面的内容。而中小企业则更看重云计算成本、易用性、可管理、易部署方面的优势。包括CRM、ERP、OA、呼叫中心、音/视频会议在内的传统ICT应用，正在越来越多地转向云计算模式。从当下的局势看，云计算必将给整个电子信息产业带来突飞猛进的发展，不但更多的中小企业将会低成本地享有IT资源，城市管理、金融、医疗、教育、地理信息等行业也将推动云计算的全面应用。

伴随着云计算产业的快速发展，大量云计算领域的专业书籍自然层出不穷。从国内市场上同类书的情况来看，有关云计算的书籍品目繁多，涉及的内容也几乎囊括了技术架构、应用案例、商业模式以及运维管理等方方面面。但从市场上大部分的书籍内容上来看，现有书籍的内容各有侧重，或重技术而轻应用，或重应用案例而轻运维管理，再或是重技术基础而轻商业模式，所述内容较为单一。而本书从云计算基本概念入手，随之介绍云计算的商业模式，理论介绍逐步深入，将读者引入到云计算的实现技术方面，内容全面丰富、覆盖面广，涉及了虚拟化、数据通信、网络与运维管理、安全与认证管理等各种主流的云计算关键技术。此外，本书还通过引入开源案例（基于OpenStack）研究，拓展介绍了完整的云计算生命周期管理。本书尤其适合作为云计算方面的研究生课程参考教材。

本书结构严谨，由浅入深、归纳条理清晰、探讨系统深入，符合读者的一般习惯，书中全面系统地介绍了云计算、云计算技术及其商业趋势，描述了它的系统架构、安全性能、应用情况、商业运营以及发展前景，讨论了云计算的部署及商业运维的多种解决方案。本书共分为7章。第1章是引言。第2章是云计算业务，介绍了基于虚拟化和云的IT行业变革、云相关的商业模型以及如何向网络运营商中引入云技术。第3章是CPU虚拟化，主要介绍了虚拟化的动机与背景、计算机的架构以及虚拟化和系统管理。第4章是数据网络——云的神经系统，主要介绍了OSI参考模型、IP协议族、IP网络QoS、广域网的虚拟化技术、软件定义网络以及IP安全。第5章是网络设备，主要介绍了域名系统、防火墙、NAT盒以及负载均衡器。第6章是现代数据中心的云存储与结构，主要介绍了数据中心基础内容和存储相关事宜。第7章是云内部的运营、管理与业务流程编排，主要介绍了企业内编排、网络与运营管理、云的编排与管理以及认证与访问管理。

本书主要由郎为民、王大鹏、陈红、姚晋芳翻译，国防科技大学信息通信学院的瞿连政、张锋军、张丽红、陈亮、王昊、张国峰、毛炳文、邹祥福、陈屹、徐延军、刘素清、陈于平参与了本书部分章节的翻译工作，蔡理金、高泳洪、王会涛、李官敏、陈林对本书的全部图表进行了整理并参与部分内容翻译，李建军、靳焰、王逢东、任殿龙、孙月光、孙少兰、马

同兵对译稿的初稿进行了审校并参与部分内容翻译，并更正了不少错误，在此一并向他们表示衷心的感谢。同时，本书是译者在忠实于原书的基础上翻译而成的，书中的意见和观点并不代表译者本人及所在单位的意见和观点。

由于云计算还在不断完善和深化发展之中，加之译者水平有限，翻译时间仓促，因而本书翻译中的错漏之处在所难免，恳请各位专家和读者不吝指正。

郎为民

目　录

译者序

第1章 »

引　言

“如果17世纪和18世纪早期是钟表的时代，18世纪末和19世纪是蒸汽机的时代，那么现在则是通信和控制的时代。”

诺伯特·维纳（摘自1948年版的《控制论——关于在动物和机器中控制和通信的科学》）

不幸的是，我们不记得我们所要描述事件的确切日期，只记得它发生在1994年秋天的某个时候。那时，宾夕法尼亚大学的诺亚·普里维斯教授在贝尔实验室做了一次令人难忘的邀请报告，本书的两位作者㊀也出席了这次报告。这次报告的重点是，建议AT&T（当时贝尔实验室是它的一部分）除了提供电信服务之外，还应通过运营数据中心，向其他的公司提供计算服务。“他们只需要连上他们的终端，就可以获得IT服务。他们只需要支付一定的费用，就可以摆脱自己运营机器、升级软件等各种运营与维护所带来的问题”。

本书中谈及多次的普里维斯教授，以他在贝尔实验室软件领域的远见卓识而闻名，而且他还是一家成功的软件公司——计算机指挥与控制的创始人兼首席执行官，他的提议对研究人员来说是较为超前的。当时，AT&T的核心业务是电信业务。AT&T的主要企业客户需要购买用户端设备（例如，专用的分支交换机和呼叫中心支持软件运行所需的机器）。也就是说，企业需要购买设备自己运行，而不是将其外包给网络提供商！

大多与会者虽然看到了这一想法的优点，但无法立即将其与他们的日常工作联系在一起，更重要的是，他们无法将其与公司所确定的经营计划联系在一起。而且，当时贝尔实验室正在将他们的计算环境从大型机和Sun工作站托管的UNIX编程环境迁移到支持微软办公软件的个人计算机上。虽然我们不喜欢这种变化（因为我们是伴随着UNIX操作系统成长起来的人），但是我们被告知，就办公信息技术而言，这就是行业发展的方向。但如果这样，那么企业将向完全相反的方向发展——把计算机放在每名员工的手中。普里维斯教授并没有否定个人计算机的发展步伐；而是更看重企业业务，那些需要在自己的工作站上维护工资数据库之类的企业用户。

在那次会议中，人们针对具体的细节问题进行了热烈的讨论。普里维斯教授援引了虚拟化和大规模并行处理技术的成果，这些足以实现他所描述的愿景。虽然，这些论据是令人信服的，但是AT&T的核心业务仍然是网络，而网络则主要集中在电信业务上。

不过，电信业务仍然由软件提供，甚至电话交换机也是由计算机控制的外围设备。在20世纪90年代，虚拟电信网络服务（例如，软件定义网络）没有与数据网络中发展的同名业务（相关内容将在本书第4章讨论）相混淆，而是出现在被称为“智能网络”上的纯软件和数据通信平台上。正是在后者的基础上，普里维斯教授认为可以提供计算服务。总之，这个想法是将数据通信和集中化的强大计算中心结合在一起，所有这些都在一个大型的电信公司的中央指挥和控制之下运行。这让当时所有与会者都对此产生了浓厚的兴趣。

将计算机作为公用设施的想法并不新鲜，道格拉斯·帕克希尔（Douglas F. Parkhill）在其1966年的书[1]中就提到了这一点。

不过最后，我们都没能把这一想法呈递给公司的高级管理层。1994年电信业所经历的时代最有可能被认为是“有趣的”，AT&T的业绩并不好，原因很多㊁。尽管贝尔实验室处于所有相关技术发展的

㊀ Igor Faynberg和Hui-lan Lu当时是贝尔实验室41领域（架构领域）技术团队的成员。

㊁ 由于地方贝尔运营公司和其他本地有商务往来的运营商也开始在服务市场上与AT&T展开竞争，所以他们不愿意从AT&T集团的网络系统部门（该部门是AT&T下属的一家生产性的单位）购买设备。

前沿，但将这些技术推荐给企业则是另一个问题，特别是在不景气的时期，提出彻底改变业务模式的建议。

大约1年后，AT&T宣布解体。本书的两位作者与贝尔实验室的大部分人员一起，去了后来成为朗讯科技（Lucent Technologies）的设备制造公司，10年后，与阿尔卡特合并组建阿尔卡特朗讯（Alcatel - Lucent）。

大约在同一时间，亚马逊（Amazon）推出了一项名为弹性计算云（Elastic Compute Cloud，EC2）的服务，它几乎完美地实现了普里维斯教授曾向我们描述的内容。世界各地的企业用户只需支付一定的费用后，就可以在“云”（或更准确地说，在其中一个亚马逊的数据中心）中创建虚拟化的机器并在这些机器上部署各种软件。而且，不仅如此，这些机器还是弹性的：随着用户对计算能力需求的增长，机器的能力也随之提高，直到满足用户的需求，同时增加适当的成本；当需求下降时，计算能力也随之降低，同时成本费用也随之减少。因此，企业不需要投资购买和维护计算机，而只需要为所使用的计算能力支付一定的费用，并且可以根据需要获得尽可能多的计算能力！

从哲学的角度来看：可以使用辩证法来看待计算形式的发展。如图1.1a所示，将基于大型机的计算作为论题，业界已经发展到基于个人工作站的计算——作为对立面。但是，得益于数据网络、分布式处理和软件自动化等技术的发展，这种螺旋式的发展带来了一种新形式的“云”，作为这两者的结合体，其中这种看似集中的按需计算所带来的便利性是与用户计算环境的自主性相结合的。另一种螺旋式发展（将在本书第2章中详细介绍）如图1.1b所示，它展示了公有云如何成为论题“传统IT数据中心”的对立面，邀请外包开发（通过“影子IT”和虚拟私有云）。对应的结合体是私有云，其中云将计算放回到企业内部，但是，是以一种非常新颖的形式。

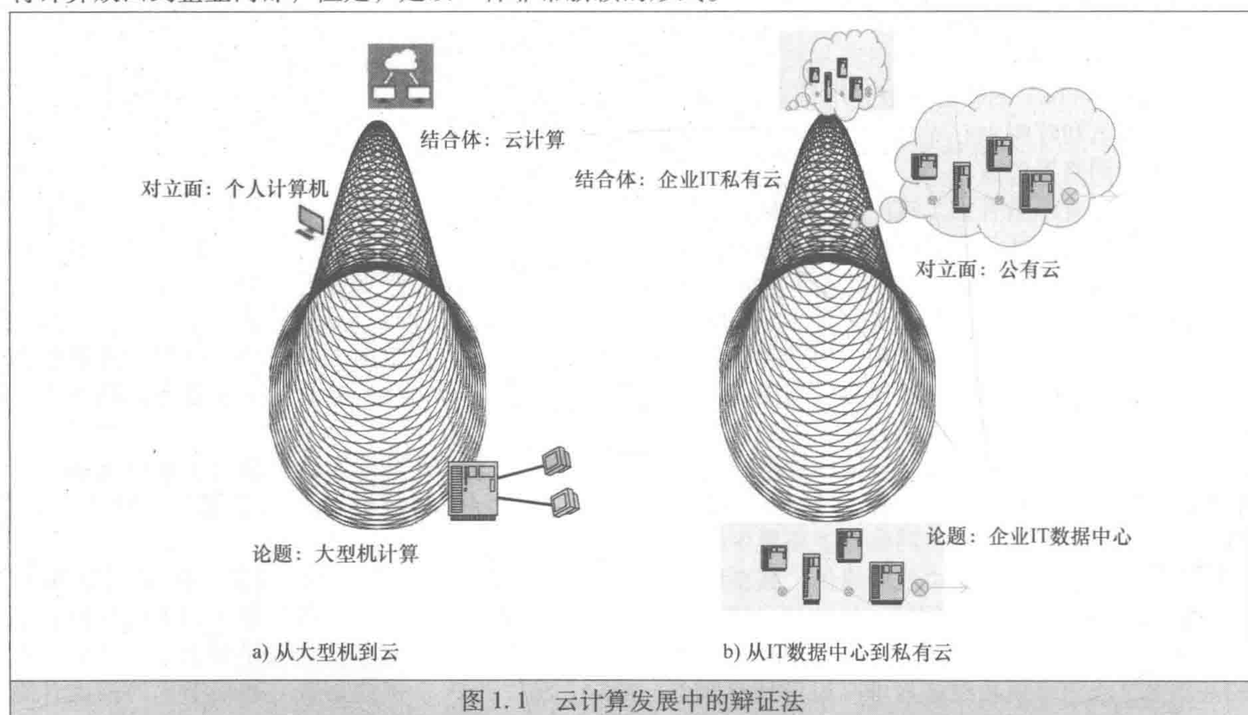


图1.1 云计算发展中的辩证法

从这个意义上，我们准备介绍一些正式的定义，这些定义已经得到了普遍的认同，从而形成了一个标准。这些定义是由美国国家标准与技术研究院（National Institute of Standards and Technology，NIST）制定和发布[2]的。首先，云计算被定义为一种模式“用于实现对可配置计算资源共享池（例如，网络、服务器、存储、应用程序和服务）进行无处不在、方便、按需的网络访问，而只需投入极少的管理工作或与服务提供商的沟通，就可以实现对这些计算资源的快速配置和发布”。这种云模式由5个基本特征、3种服务模式和4种部署模型组成。

5个基本特征如图1.2所示。



①按需自助服务。用户可以单方面配置计算能力，例如，根据需要自动配置服务器时间和网络存储，而无须与任何服务提供商进行沟通。

⑤可度量的服务。云系统可以在某种抽象程度上使用适合这种服务类型的计量能力（例如，存储、处理、带宽和活跃的用户账户），来自动地控制和优化资源的使用。资源的使用是可监、可控和可实时上报的，并且这对被使用服务的用户和提供商来说都是透明的。

②广泛的网络访问。可以在整个网络上通过标准的机制获得计算能力，可以使用各种异构的瘦客户端或胖客户端平台(例如，移动电话、平板电脑、笔记本电脑和工作站)来进行访问。

③资源池。提供商将计算资源汇集在一起通过多租户模式为多个用户提供服务，所提供的各种物理和虚拟资源会根据用户的需求动态地进行分配和再分配。这体现了一种位置无关性的感觉，也就是说，用户一般无法掌控或了解被提供的资源的确切位置，但是可以在一个较高的抽象层次（例如，国家、州或数据中心）上指定服务资源的位置。

④快速弹性的扩展。可以对计算能力进行弹性的配置和发布，在一些情况下可以契合需求自动地实现向外或向内的快速扩展。从用户的角度来看，可用的配置能力似乎是无限的，可以在任意时间占有任意数量的计算能力。

图 1.2　云计算的基本特征⊖

现在众所周知的 3 种云计算服务模式是软件即服务（Software-as-a-Service，SaaS）、平台即服务（Platform-as-a-Service，PaaS）和基础设施即服务（Infrastructure-as-a-Service，IaaS）。NIST 将其定义如下：

1）软件即服务（SaaS）。向用户提供的能力是使用提供商在云基础设施上运行的应用程序。这些应用程序可以通过各种客户端设备进行访问，这些客户端设备可以是瘦客户端接口，例如，网络浏览器（如基于 Web 的电子邮件）或应用程序接口。用户不需要管理或控制底层的云基础设施，包括网络、服务器、操作系统、存储，甚至个人应用程序功能，唯一可能的例外是需要用户设置有限的用户定制应用程序配置。

2）平台即服务（PaaS）。向用户提供的能力是在云基础设施上部署用户创建的应用程序，或者获取使用提供商支持的编程语言、库、服务和工具构建的应用程序。用户不需要管理或控制底层的云基础设施，包括网络、服务器、操作系统或存储，但是需要控制所部署的应用程序，并且可能需要设置应用程序托管环境的配置。

3）基础设施即服务（IaaS）。向用户提供的能力是配置处理、存储、网络和其他的基础计算资源，使用户能够在其上部署和运行任意的软件，这些软件可以包括操作系统和应用程序。用户不需要管理或控制底层的云基础设施，但是需要控制操作系统、存储和所部署的应用程序。并且，在选取网络组件（例如，主机防火墙）上可能需要进行有限的管控。

随着时间的推移，其他的服务模式已经出现，不过这些服务模式更多的是出现在与营销相关的文献中。但是，著名的“伯克利的云计算”[3]的作者选择“避开诸如‘X 即服务（X as a Service，XaaS）’这样的术语”，因为难以对它们之中所存在的确切差异达成一致，也就是说，对 X 所代表的某些服务的价值之间的差异难以认同。

4 种云部署模型由 NIST 定义如下：

1）私有云。这类云基础设施是由包含多个用户（例如，业务部门）的单个组织专用的。它可由该组织、第三方或它们之间的某种组合所拥有、管理和运营。并且，它可能有也可能没有特定的场所。

2）社区云。这类云基础设施是由具有共同关注项目（例如，任务、安全要求、政策和合规性注意事项）的组织内的特定的用户群体专用的。它可由该社区中的一个或多个组织、第三方或它们之间的某种组合所拥有、管理和运营。并且，它可能有也可能没有特定的场所。

⊖ 援引自 NIST SP 800－145，p. 2。

3）公有云。这类云基础设施向普通公众开放使用。它可由商业、学术或政府组织，以及它们之间的某种组合所拥有、管理和运营。它存在于云提供商的场所中。

4）混合云。这类云基础设施是由两个或多个不同的云基础设施（私有云、社区云或公有云）组成的，其中各个云基础设施依然保持独特的实体，只是按照标准化或专用的技术绑定在一起，从而实现数据和应用程序的可移植性（例如，用于云之间负载均衡的云爆发）。

云计算不是一种单一的技术，而是一种业务的发展，它的实现得益于多个学科：计算机体系结构、操作系统、数据通信以及网络和运营关联。正如将要看到的那样，最后一种学科伴随着网络已经存在了很长的时间，只不过云计算的引入将其发展自然地推向了一个新的方向，再次验证了我们从诺伯特·维纳这本书中引用的，作为本书引语的那句话。

正如本书第 2 章所述，云计算已经对信息技术行业产生了革命性的影响，电信行业也是如此。电信运营商要求供应商只提供软件，而不是“盒子”。在行业中已经有几项相关的标准化规范，或许更重要的是，已经有了一些开源软件包来构建云环境。

在标准化之前，自然而然的是大量的研究和开发方面的工作。在 2011 年，本书作者之一[⊖]在阿尔卡特朗讯成立了 CloudBand 产品部门，在贝尔实验室研究工作的帮助下，开发出了电信云平台。正是在 CloudBand 的背景下，本书三位作者见面，并产生了撰写本书的想法。

起初，我们计划将本书作为一本云计算的教科书。在史蒂文斯理工学院从事开发和教授相关方面研究课程的经历使我们了解到，即使是最聪明和最优秀的学生，在中央处理器（Central Processing Unit，CPU）虚拟化（这个课程在计算机体系结构或操作系统课程方面很少被教授），以及在数据通信中的一些特定方面，也缺乏足够的知识。网络和运营管理很少会成为现代计算机科学课程的一部分。

事实上，在业界类似的知识空白似乎是普遍存在的，因为行业内的工程师往往被要求具有很强的专业性，专业划分的程度太细，因此我们希望本书可以通过提供一个总体的多学科基础来弥补这一空白。

本书其余部分的结构如下：

1）第 2 章主要介绍“是什么”而不是“怎样做”。给出了一些定义，并通过一个网络功能虚拟化的特殊案例研究描述了一些相关的业务考量。另外，介绍有关云计算的整体框图。至于“怎样做”，则是以后几章将要介绍的内容。

2）第 3 章解释 CPU 虚拟化的原理。

3）第 4 章专门介绍网络——云的神经系统。

4）第 5 章介绍网络设备、云数据中心和专用网络的组成部分。

5）第 6 章介绍现代数据中心的整体架构及其组成部分。

6）第 7 章审视云中的运营和管理，并通过对 OpenStack 的案例研究，阐明业务流程、身份认证和访问管理的概念，其中所研究的案例——OpenStack，是一个流行的开源云项目。

7）附录部分对前面讨论的主题细节进行深入的研究。

参考文献部分（也是各自主题的参考书目）分列在各个章节的后面。

从前面所列的这本书的大纲中，我们应当注意到，有 3 个基本的主题没有设置专门的章节进行介绍。而是，将其放在其他章节内讨论，因为它们涉及了该章节的主题。

其中一个主题是安全性问题。毫无疑问，这是唯一可能导致云计算遭破坏的问题。安全性问题存在很多方面，所以我们认为应该在各相关章节内讨论并解决与各章节有关的安全性方面的问题。

另一个没有“集中”介绍的主题是标准化问题。同样，我们在讨论具体技术主题的同时介绍相关标准和开源项目。第三个主题是历史问题。在工程学中众所周知的是，很多现有的技术解决方案并不是最优的，它们的存在得益于它们的历史发展。在一个学科课程的讲授过程中，最重要的是要将这类问题指出来，我们已经尽力做到了这一点。同样，我们将在所要介绍的每一个技术背景中，指出这一问题。

⊖ 多尔·斯库勒当时是阿尔卡特朗讯的副总裁兼 CloudBand 产品部门的总经理。



参考文献

[1] Parkhill, D. F. (1966) Challenge of the Computer Utility. Addison - Wesley, Reading, MA.
[2] Mell, P. and Grance, T. (2011) . Special Publication 800 - 145: The NIST Definition of Cloud Computing. Recommendations of the National Institute of Standards and Technology. US Department of Commerce, Gaithersburg, MD, September, 2011.
[3] Armbrust, M. , Fox, A. , Griffith, R. , et al. (2009) Above the Clouds: A Berkeley view of Cloud Computing. Electrical Engineering and Computer Sciences Technical Report No. UCB/EECS - 2009 - 2A, University of California at Berkeley, Berkeley, CA, February, 2009.

第 2 章 »

云计算业务

在本章中，将对云计算的商业影响进行评估。

首先从概括 IT 行业转型的过程开始，其发展历程跨越的阶段较少。首先是虚拟化，其次是向云转型。正如人们看到的那样，这一过程是以辩证螺旋的形式发展的，其中受到不同发展方向相互冲突所带来的影响。行业发展趋势正将计算与企业相分离，从而诞生“影子 IT”和虚拟私有云。最终，这一发展又会以私有云的形式将计算带回到转型的企业 IT 领域中。

接下来，不谈企业，而是只考虑电信业务，同样这类业务也经历了类似的过程，即众所周知的网络功能虚拟化（Network Functions Virtualization，NFV）。如今，这种网络功能虚拟化正在发展出它自己的私有云（本书的作者都已参与了这一过程）。

当然，云转型必然会影响到其他的业务部门，而本书的目的随着本书内容的不断完善，也表明了此时已经把握到了云对行业影响的这条主线。正如适用于物理场（例如力学）的数学方程，同样可以适用于其他领域（例如电磁场）一样，通用的行业模式也必然适用于各种行业。云的影响将在许多其他行业中被人们看到和感受到。

2.1 基于虚拟化和云的 IT 行业转型

在过去的 10 年中，IT 行业已经经历了大规模的转型，这对业务和行业方面的新应用和服务的引入产生了巨大的影响。要想了解发生了什么，先来看一下以前的情况。

云计算之前，开发基于软件的产品和服务需要在前期投入高昂的资金，投资失败的风险很高，产品入市的进度缓慢，而且还需要在基础设施的运营和维护方面持续投入运营成本。开发人员通常负责整个系统的设计和实施：从物理基础设施（例如，服务器、交换机、存储器等）到软件可靠性基础架构（例如，集群、高可靠性和监控机制）和通信链路的选择——也就是将业务逻辑转换为应用程序的所有方面。要将特定服务的应用部署在专用的基础设施上，还要为每一个服务制定单独的容量和能力规划。

以一个实际的情况为例。2000 年，本书的一位作者[㊀]创建了一家名为“Zing 互动媒体”的公司，该公司的业务是让广播听众通过简单的语音指令与收音机上听到的内容进行交互。想想在收音机上听到一首非常棒的歌曲，或是你感兴趣的广告时，设想一下，如何用简单的语音指令让你能够订购这首歌或同广告商进行互动。在当今世界，这可以作为一种基于云 SaaS 解决方案实现的经典案例。

但是在 2000 年，作者的公司不得不做很多事情来创建这个服务。首先，当然是开发实际的产品来提供服务。但在这之前，最重要的还是需要投入大量的资金，而这些对最终用户来说是不可见的[㊁]：

1）在托管网站上租用空间［在这种情况下，在 AT&T 托管设施上租用了一个安全的空间（俗称一个“笼子”）］。

2）预测峰值使用量并为服务开发冗余方案。

3）指定满足此容量和功能计划所需的服务器的技术要求（其中涉及大量的购物功能）。

4）协商供应商和支持合同，并购买和安装足够的服务器以满足该容量和功能计划（其中有些服务

㊀ Dor Skuler。

㊁ 这些行动对于后来转为 SaaS 的所有产品来说都是典型的应用情况。

器将不可避免地处于空闲状态）。

5）租用专用 T1[㊀]线路连接到我们的“笼子”，并支付其全部的容量费用，无论实际是否使用和具体使用了多少。

6）购买网络设备（交换机、电缆等）并将其安装在我们的“笼子”中。

7）在服务器上购买并安装软件（操作系统、数据库等）。

8）购买和安装负载均衡器、防火墙和其他网络设备[㊁]。

9）聘用一个由网络专业人员、系统管理员、数据库管理员等组成的 IT 团队来维护这套设施。

10）部署和维护其中唯一用来实际提供 Zing 交互媒体服务的软件。

需要注意的是，这笔投资具有巨大的前期成本。这是在启动服务之前需要投入的，并且与产品没有直接的关系。出于必要性，这笔投资是以高峰使用模式考虑的，最少也是以中等程度的使用情况进行考虑的。即使采取了所有的预防措施，这类投资也是属于基于知识性的猜测。此外，随着服务的成功，扩展系统需要规划和较长的交货时间：购买的服务器需要送货时间，访问托管网站需要规划和批准，并且，网络提供商需要几周的时间才能激活新订购的通信链路。

稍后再回到这个例子，来描述今天如何使用云部署我们的服务。

这个例子代表了企业 IT 部门在部署服务（例如，电子邮件、虚拟专用网络或企业资源规划系统）时必须处理的问题。事实上，很多大型公司的开发部门也都面临着同样的问题。

当开始一个新项目时，开发经理需要遵循以下步骤：

1）进行总体成本估算（存在许多不确定性）。

2）批准预算和空间以托管服务器和其他设备。

3）撰写新硬件的购买请求。

4）通过采购部门购买服务器（可能需要 3 个月左右的时间）。

5）雇佣支持团队直到服务器安装和配置结束、安全策略部署完毕，以及最后启用连接。

6）安装操作系统和其他软件。

7）开始开发实际的增值软件。

8）当需要额外的设备或外部软件时返回步骤 1）。

当需要测试时，该过程会随着每一个测试专用系统的数量呈指数规模增长。一个典型的例子：当软件产品需要进行大规模的压力测试时，整个基础设施必须到位，等待测试，这可能只在一个星期甚至一个月内仅仅运行几个小时。

反过来，看一下如何使用带有私有云设置的云以及所谓的“影子 IT”来解决类似的问题。

需要注意的是，如今，上述过程已经得到了精简，从而使开发人员和服务提供商只需要关注他们所创造的增值部分。这种转变的实现得益于 IT 转型成为一种新的做事方式。两大主要的推动力相继来到：第一是虚拟化，第二是云本身。

虚拟化（在下一章将进行详细介绍）实际上已经出现很多年了，只是最近才被 IT 经理们“重新发现”，他们希望借此来降低成本。简单来说，虚拟化是通过硬件的复用来整合计算。例如，如果一家公司拥有 10 台硬件服务器，每台运行自己的操作系统和一个 CPU 利用率相当低的应用程序，那么虚拟化技术将可以使用一个或两个强大的服务器来替换这 10 台服务器，而不需要让软件产生任何的变化或对高性能的发挥造成任何的影响。在下一章将看到，虚拟化的关键是虚拟机管理程序，它模拟了硬件环境，以便运行在它上面的每个操作系统和应用程序“认为”它们是在自己的服务器上运行。

因此，可以将运行在未被充分利用的专用物理服务器[㊂]上的应用程序逐渐转移到虚拟化的环境中，首先是服务器整合。因此，只需要购买和维护较少的服务器即可，从而节省资本支出（Capital Expenditure，CapEx）和运营支出（Operational Expenditure，OpEx）。考虑到在过去典型 IT 预算中有 2/3 将被用于维护，因此可以认为这是一项重大的成就。其他的好处还包括对可用性、灾难恢复和灵活性的改进

㊀ T1 是美国的一种高数据速率（1.544 Mbit/s）传输业务，可从电信运营商处租用。它基于最初由贝尔实验室开发的 T 载波系统，并部署在北美和日本。欧洲基于此推出了后续 E 载波系统（即 E1 服务），欧洲的 E1 业务可以提供2.048Mbit/s 的数据速率。

㊁ 将在第 5 章讨论网络设备。

㊂ 如果服务器上的应用程序在典型的 X86 处理器上的平均利用率为 5% ~10%，则人们认为该服务器是未充分利用的。

（因为部署虚拟服务器要比部署物理服务器快得多）。

这些改变为服务提供商带来了极大的收益，而对 IT 服务的消费者来说则和以前的体验在很大程度上没有什么不同——因为刚才描述的虚拟化是静态的。运行的服务器越少，利用率越高。虽然这是可以确定的重要一步，但是它并没有改变消费计算资源的基本复杂性。

云是向前迈出的重要一步。云提供给 IT 行业的是能够转向到具有最少的前期投资和最低风险的“服务为中心”的“按需付费”业务模式。开发新应用的个人和企业可以从低成本的基础设施实际上无限的规模中受益，从而允许用户只为实际使用的资源付费。此外，云计算基础设施“抽象化”，可以让用户将 100% 的精力全部花费在应用程序的开发上，而不是设置和维护一般的基础设施。像亚马逊和 Google 这样的公司已经建立了大规模、高效率的云服务。

正如在上一章中看到的那样，从基础设施的角度来看，云已经推出了一种多租户（在同一物理基础设施上为许多用户提供支持）、弹性的、配有可编程接口（通过 API）、全自动化和具有自我维护能力的平台，而且更重要的是，这一平台具有非常低的总体拥有成本。起初，云平台提供了计算和存储等基本的基础设施服务。近年来，云服务已经升级到软件产品实施中，以提供越来越多的通用服务，例如负载均衡即服务或数据库即服务，这使得用户可以更加专注于他们的应用核心功能。

这里以一个例子来说明这一点。最初，云用户只能创建一个虚拟机。如果该用户需要数据库，则必须亲自购买、安装和维护数据库。这里的一个微妙的问题是许可证——通常，购买的软件许可证会和有限数量的物理机器进行绑定。因此，当虚拟机被转移到另一个物理主机时，该软件可能不会运行。然而，随着数据库即服务的出现，用户只需要选择所选的数据库并开始使用它即可。获取数据库软件和相应许可证，以及软件的安装和维护的任务，现在都是由云提供商来承担的。类似地，为了实现负载平衡（在引入负载均衡即服务之前），用户需要创建和维护要平衡的服务器的虚拟机以及负载均衡器本身。正如将在第 7 章和附录中看到的那样，当前技术和云服务只需要用户指定服务器即可，在需要的时候这可由云提供商复制，引入负载均衡器以平衡副本。

云的最新发展推动了对应用程序生命周期管理的支持，提供了通用服务来替代那些应用程序本身必须具备的服务。此类服务的例子包括自动部署、自动规模缩放、应用程序监控和自动修复。

例如，在过去，应用程序开发人员必须创建监控工具作为应用程序的一部分，然后创建一个算法来决定是否添加更多的容量。如果是这样，这些工具将需要设置、配置和上线新的虚拟机和可能的负载均衡器。类似地，这些工具需要判定应用程序是否正常，如果不是，则通过创建新的服务器、加载保存的状态并关闭失败的服务器来启动自动加密。

使用新的生命周期服务，所有应用程序开发人员现在需要做的只是宣布做出这样的决策规则，并让云提供商的软件执行必要的操作即可。再次，开发人员的精力只需要（也只能）集中在应用程序本身的功能上。

这背后的技术是，云提供商基本上已经创建了通用服务，并为每个服务提供了适当的应用程序编程接口（Application Programmer's Interface，API）。实际的情况是，所有应用程序中的共同功能已经被“抽象化”了，也就是将它们以“构建模块”的形式提供。这种类型的模块化一直是软件开发的原则，但是以前只能通过对本地库进行严格指定的过程调用才能实现，而如今可以以一种高度分布的方式来完成，构建模块驻留在需要组装它们的应用程序之外的机器上。

图 2.1 说明了行业中的这一问题。在云出现之前，实际的增值应用程序只是最终用户所看到的冰山一角，而在用户看不到的、较大的、不可见的部分中，仍然需要大量的投入。

在行业中，反映这一变化的具有代表性的例子是 Instagram。Facebook 花费 10 亿美元收购了 Instagram。在收购时，Instagram 只有 11 名员工，却管理着 3000 万个用户。Instagram 没有物理基础设施，只有 3 个人被雇佣来管理亚马逊云中的基础设施。不需要资本支出，不需要采购和维护物理服务器，也不需要为管理它们的技术人员支付费用，等等。这使得公司在两年内产生了 10 亿美元的价值，对人力或基础设施的投资很少，甚至没有。大部分的公司花费都用在了用户的获取和保留方面。随着 Instagram 的用户越来越多，云将允许 Instagram 自动扩展，服务能力不断增长，而不会出现服务暂停或中断。

回到早期的“Zing 交互媒体”的例子上——如果这项服务在今天被推出，它一定会遵循 Instagram 的模式。没有必要租用“笼子”，购买服务器，租用 T1 线路，或者经历其他上述的限制。取而代之的是，我们只需要专注于交互式广播应用程序本身即可。此外，也不需要雇佣数据库管理员，因为我们



的应用程序可能会使用数据库即服务功能。最后，只需要雇佣较少的开发人员，因为开发这样一个强大、可扩展的应用程序将与在云提供商的相关服务中定义生命周期管理规则一样简单。

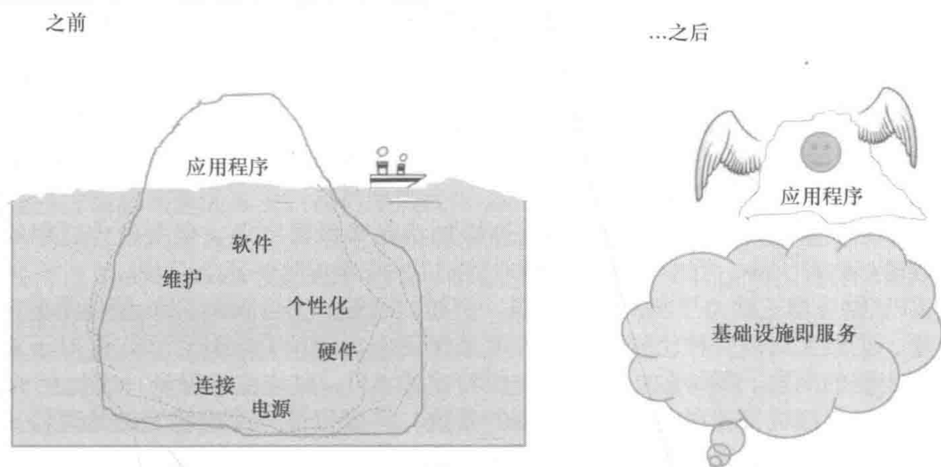


图 2.1 应用程序部署中的投资——之前和之后

在公司软件开发方面，存在两个趋势：影子 IT 和私有云。

借助影子 IT 的趋势，内部开发人员在面对是选择遵从上述过程（与虚拟化相比没有太大变化）还是使用云服务时，通常会选择绕过 IT 部门，拿出信用卡并开始在公共云上开发。考虑一下前面讨论的压力测试的例子——通过相对简单的逻辑，开发人员可以在需要时以非常大的规模运行此测试，并且只需要为实际的使用进行付费。如果需要扩大规模，只需要一个简单的更改，就可以立即实现。回顾之前传统流程中的步骤及其相关的成本（时间和经济方面），这种方法能够脱颖而出的原因也就自然变得显而易见。

许多首席信息官（Chief Information Officer，CIO）已经察觉到了这一趋势，并且认为仅在他们的数据中心（通常被称为私有云，但实际上并非如此）实施虚拟化仍是不够的。影子 IT 的风险很大，其中就包括对人员管理的失控。由于重要的公司数据被复制到云中，所以也存在着重大的安全隐患。对关键数据（将在附录中详细介绍）的访问问题尤为重要，因为这常常涉及隐私，并受到法规和法律的约束。例如，美国健康保险便携性和责任法案（US Health Insurance Portability and Accountability Act，HIPAA）有严格的隐私规则，公司必须遵守这些规则。在保护数据访问的规则方面，还有一个重要的例子是美国法律，被称为“萨班斯－奥克斯利法案（Sarbanes－Oxley Act，SOX）”，其规定了所有美国上市公司董事会和会计师事务所的标准。

在影子 IT 的威胁下，这些考虑导致 CIO 采取了新的方法。一个是被称为虚拟私有云的方法，这是通过从云提供商那获得一个安全区域（一组专用的资源）来实现的。这种方法可以让公司以受控的方式享受云带来的所有好处，公司可以完全控制 IT 部分的安全性和成本。服务级协议和潜在的责任在此得到明确的界定。

第二种方法是在公司自己的数据中心建立真正的私有云。这种方法的实现技术发展很快，所以供应商已经开始在软件产品中提供云的全部功能。本书将在第 7 章和附录中详细讨论的一个例子是，开发支持云软件的开源项目——OpenStack。借助这样的产品，企业 IT 部门就可以推出自己的数据中心，从支持虚拟化到建立真正的云，其服务与云提供商提供的服务类似。这些私有云在企业中提供内部服务，具有公共云的大部分优势（虽然规模有限），但由于云提供商的利润被消除，因此实现了全面控制并最终降低了成本。

技术公司的发展趋势是从公共云开始的，然后在规模壮大后，转向真正的私有云来节省成本。最著名的例子莫过于 Zynga，这是一家游戏公司，开发了 Farmville 等游戏。Zynga 从亚马逊开始，提供它的网络服务。当游戏开始起步，其使用模式变得可预测时，Zynga 将其转移到被称为 zCloud 的内部云上，并针对游戏需求进行了优化。类似地，eBay 也已经在 7000 台服务器上部署了 OpenStack 软件，目前这款软件已经拥有了 95% 的市场份额。

由此可见，云的好处是相当明显的。但云也有缺点。

前面，本书已经讨论了一些安全问题（将在整本书中着手解决这些安全问题）。我们很容易爱上云所带来的简单性，但安全挑战同样是非常真实的，在我们看来，这些问题不容低估。

另一个问题是控制硬件选择以满足可靠性和性能要求。从心理学角度而言，让开发人员放弃对所需服务器的准确规格的控制，并放弃对哪个 CPU、内存、外形尺寸和网卡的选择使用并不容易。其实，这并不仅仅是心理上的问题。要让开发人员在确定满足规范之前，只能相信云基础设施能够正确响应 API 调用，达到提供计算能力的目的，还是比较难的。在这种情况下，开发和评估总体软件模型以支持高度可靠和高性能的服务，尤为重要。

正如将在本书后面看到的那样，云提供商通过添加功能来预留特定（但是硬件通用的）配置参数（如 CPU 内核数量、内存大小、存储容量和网络“管道”）来响应这一点。

英特尔等 CPU 供应商正致力于解决这些问题。例如，需要一个可预测 CPU 使用量的应用程序。直到最近，在云中，仍然无法以一种比较精细的粒度来保证应用程序会收到什么，这对于实时应用程序来说可能是一个主要的问题。英特尔正在通过提供特定的 API，允许主机保证一定比例的 CPU 分配给特定的虚拟机。通过虚拟机管理程序和云提供商的系统，将虚拟机分配给特定的处理器或一系列进程（即所谓的 CPU pinning）来实现这种功能，供用程序所使用。

使用较高的抽象层次，可以获得简单性，但是由于使用了通用服务，所以个性化的定制功能将非常有限。否则，如果不对外提供 API，那么就无法使用这些功能。例如，如果想要使用指定供应商负载均衡器的特定高级功能，那么将无法在通用云中实现。而是只能使用云提供商通过 API 对外开放的负载均衡功能，在大多数情况下，甚至不知道哪个供应商能够支持这项服务。

这里给出的解决方案是降低抽象级。以最后一段为例，可以购买供应商负载均衡器的虚拟版本，将其作为虚拟机，作为项目的一部分，然后使用它。换句话说，较高的抽象层次可能无法满足独特的需求。

2.2 云端业务模式

云服务提供商，如 Google 或亚马逊，运行着庞大的基础设施。据估计，Google 拥有超过 100 万台物理服务提器，而亚马逊云正在为 150 万 ~200 万台虚拟机提供基础设施。这些巨大的数据中心是使用高度商品化的硬件构建的，拥有非常小的运营团队（团队中只有数十人在管理着所有的 Google 服务器），利用自动化来提供新的运营效率水平。虽然基础设施组件本身并不是高度可靠的（亚马逊只提供 99.95% 的 SLA），但以基础设施自动化和应用程序编程的方式来利用这一基础架构，就可以提供相当可靠的服务（例如，Google 搜索引擎或 Facebook Wall），而其成本只有其他行业类似服务费用的一小部分。

云提供了新的基础设施效率和业务敏捷性水平，并通过新的运营模式（例如，自动化、自助服务、标准化商品要素）来实现，而不是通过基础设施要素的性能优化来实现。CapEx 对硬件的投资不到这些基础设施总拥有成本的 20%，其余的主要是运营和许可成本。云操作模式和软件选择（例如，使用开源软件）可以显著降低总体成本——而不仅仅是在硬件上，就像虚拟化一样。

让我们简单看一下云提供商与软件和服务供应商提供的业务模式，分别在后面的小节中介绍。

2.2.1 云提供商 ★★★

云为其服务提供了一种实用模型：计算、存储、应用程序和操作。它提供了一系列的定价模式，平衡了终端用户的灵活性和价格。灵活性越高价格越高——无须任何承诺即可随时随地满足用户需求。针对预留的容量制定更好的价格，或者在特定的时间内保证提供一定的使用量，这可以让云提供商更好地规划其容量。例如，在撰写本章时，在亚马逊的网站上使用它的定价工具，了解到在 AWS 中针对中型机器的报价为每小时 0.07 美元，用于按需使用。相同机器预留容量的报价为 0.026 美元，折扣 63%。此定价不包括网络、数据传输或其他费用[㊀]。

㊀ 该引用价格是在 2015 年 1 月 20 日获得的。

更高的价格是针对特殊服务收取的，例如前面提到的虚拟私有云。最后，最优惠的价格是现货定价，即云提供商定义何时提供所寻求的服务（即在供应商的能力预计未得到充分利用的时候），这是离线计算任务的绝佳选择。对于云提供商来说，这可以确保实现更高的资源利用率。

亚马逊 AWS 引领了一个有趣的趋势是价格的不断下降。随着亚马逊规模不断扩大，以及存储和其他成本的下降，目前亚马逊正采取持续降低定价的方式，提高它的竞争优势，并为潜在的客户提供更多更有吸引力的案例。此外，亚马逊还不断增加创新服务，例如上述提到的更高层次的应用程序抽象，当然这些服务会带来新的费用。额外的费用主要用在网络、配置更改和特殊的机器类型需求等方面。

对于对云业务方面感兴趣的人士，我们强烈推荐 Joe Weinman 的相关著作[1]，该书还提供了一个有用的且非常有趣的网站，其中包含了一套用于处理实用程序和云计算的结构、动态和财务分析的模拟工具。我们还推荐由 Timothy Chou 发表的一篇关于云业务的论文[2]，重点介绍了软件的业务模式。

2.2.2 软件和服务供应商 ★★★

要构建私有云，CIO（Chief Information Officer，首席信息官）需要创建一个包含物理服务器、存储等设施的数据中心㊀然后，为了将其转换为云，可以选择从专有供应商（如 VMware）购买基础设施软件或使用开源软件。OpenStack（将在本书第 7 章做进一步介绍）是一个开源项目，它可以让用户构建一个与亚马逊 AWS 提供的服务类似的云服务。

尽管开源项目提供的软件是免费的，但是在具体的实践中，当涉及大型的开源项目时，仍然需要承担与维护相关的费用。因此，大多数的公司不喜欢直接从开源库中获取软件，而更愿意从提供支持和维护（如升级、bug 修复等）服务的供应商处购买软件。像 Red Hat 和 Canonical 这样的公司领导着这一领域。这些系统的定价通常基于云端群集中使用 CPU 插槽的数量。通常，费用是按年收取的年度费用，不取决于实际使用的指标。

此外，大多数的公司使用专业服务公司帮助他们建立（并经常管理）他们的云环境。这通常是根据在每个项目上所花费的时间和物质的基础上制定价格的。

2.3 将云带到网络运营商

处在云演进最前沿的是电信基础设施的转型。正如前面提到的，电信提供商（他们通常在其各自国家中也受到相关部门的监管）提供了迄今为止最可靠和最安全的实时业务。100 多年来，电信设备已经从机电交叉连接电话交换机发展到高度专业化的数字交换机，数据交换机使现有的电信网络成为可能。此外，这些交换机已经与运行操作和管理软件的专用网络设备㊁和通用高性能计算机互联。

网络功能虚拟化（Network Functions Virtualization，NFV）是运用云原理，彻底改变“基于硬件盒”的电信世界的一种新的发展产物㊂。

首先来解决网络运营商想要解决的问题。虽然今天所知道的“网络功能”大部分是由软件提供的，但这类软件都是运行在专用的“电信级”硬件设备之上的。“电信级”意味着硬件是：①专门被设计用在电信网络中运行；②在网络中的设计寿命达到 15 年以上；③99.999%（“五个九”的级别）的时间内功能无异常（即每年只能出现不超过 5min 的停机时间）。这就使定制设备的安装和维护成本异常高。尤其是考虑到“摩尔”定律，即计算机能力每 18 个月增加一倍，因此可以很容易想象到对专用硬件设备给出 15 年服务承诺所带来的问题。

随着竞争的加剧，网络提供商一直在寻求一种解决方案，以应对利润的减少和不断增长的竞争。而目前这场竞赛已经不仅仅来自于电信行业内部，还涉及网络服务提供商，即所谓的 Over – The – Top（OTT）。

解决这个问题需要一种新的运营模式，它可以降低成本，加快推出新服务以实现收入的增长。

为了解决这个问题，7 家世界领先的电信网络运营商联手合作，制定了一套标准，成为发展虚拟化

㊀ 第 6 章讨论了数据中心的结构。

㊁ 将在第 5 章进行描述。

㊂ 为了充分披露，从他们的简短提要中可以推断，该提要的作者应该是这个领域的第一批行动者，因此他们的看法自然非常乐观。

网络服务的框架。2012 年 10 月 12 日，全球 12 家网络运营商㊀的代表发布了白皮书，阐述了这样做的好处和挑战，并发出了参与这项行动的呼吁。

不久之后，其他 52 家运营商以及电信设备、IT 供应商和技术顾问组成了 ETSI NFV 行业规范组（Industry Specifications Group，ISG）。

需要采取措施的方面可概括如下。

第一，改善运营。运行包括来自多个供应商设备的网络太复杂，需要太多的开销（与运营商相比，电信运营商必须处理更高数量级的备件数量）。

第二，降低成本。使用自动化管理和维护基础设施将只需要 1/10 的人员参与“手动”操作。因此，电信网络中的“硬件盒”数量将比云运营商中的约大 10000 倍。

第三，精简高接触流程。目前，配置和扩展服务需要手动干预，并且扩展现有服务需要 9 ~ 18 个月的时间，而云则可以承诺即时缩放。

第四，缩短开发时间。推出新服务需要 16 ~ 25 个月的时间。将其与 IT 行业的数周进行比较，并与云中的立即实施服务实例化相对比，何优何劣可见一斑。

第五，降低更换成本。随着服务寿命的不断缩短，还需要更换与硬件配套的软件，其中针对硬件方面的措施将在第六条（也是最后一条）中给出。

第六，降低设备成本。将专有供应商专用硬件的价格与基于 x86 的商品现货服务器的价格进行比较，并从中得到启示。

为了应对上述问题，需要采用行之有效的虚拟化和云原理。为此，NFV 就是将之前讨论过的许多相同的云原理融入到电信领域。首先，虚拟化与路由、语音通信、内容分发等相关的网络功能相结合，然后将其运行在大规模、高效的云平台之上。

NFV 空间可以分为两部分：NFV 平台和运行在其上的网络功能。其想法是将网络功能运行在嵌入网络中的通用共享平台（NFV 平台）上。当然，网络是一般云和 NFV 之间的主要区别，因为后者的存在理由是能够提供基于网络的服务。

NFV 是指用虚拟化替代物理部署，动态部署网络功能，通过现有通用（Common Off - The - Shelf，COTS）的硬件实现网络需求。NFV 平台能够实现云节点安装和操作的自动化，协调大规模的分布式数据中心，管理和实现应用程序生命周期自动化，并利用网络。毋庸赘言，该平台对所有厂商都是开放的。

要想了解有关 NFV 动态方面的问题，请参考内容分发网络（Content Delivery Networking，CDN）服务（其所有方面都在专著[3]中进行了深入的讨论，为此这里强烈推荐）。简而言之，当内容提供商（例如，电影流媒体网站）需要通过互联网提供实时服务时，带宽成本（和拥塞）是一个障碍。针对这一问题，一种有效的解决方案是在很多服务器上复制内容，这些服务器遍布于运营商网络中的各个地理位置上，以满足各本地用户的需求，同时收取一定的费用。目前，这意味着部署和管理物理服务器，这些服务器带有前面所讨论的一些问题。一个问题是，需求通常是基于一天中的时间。由于在美国东海岸观看电影的时间与日本不同，因此相应的服务器将被交替利用，并且无法长时间地持续提供服务。将 CDN 服务器动态部署到需要服务的用户附近的数据中心的这种能力，是一种明显的优势，这不仅节省了成本，而且内容提供商和运营商提供了前所未有的灵活性。

类似地，虽然专业性较强，但是能够立即从 NFV 中受益的电信应用的例子是第三代（Third Generation，3G）[4]移动通信的 IP 多媒体子系统（IP Multimedia Subsystem，IMS）和第四代（Fourth Generation，4G）宽带无线服务[5]的演进分组核心（Evolved Packet Core，EPC）。作为一个简单的例子：应当考虑部署的灵活性——在涉及的网络提供商之间——支持漫游的网络功能㊁）。

网络提供商认为 NFV 同时具有颠覆性和挑战性，在这个领域的很多网络供应商也是如此。

可以将制定 NFV 解决方案的原则概括为以下几点：

㊀ AT&T（美国电话电报公司）、BT（英国电信）、CenturyLink（世纪电信）、中国移动、Colt、Deutsche Telekom（德国电信）、KDDI、NTT（日本电报电话公司）、Telecom Italia（意大利电信）、Telefonica（西班牙电信）、Telstra（澳洲电信）和 Verizon（威瑞森公司）。

㊁ 如 IMS 中的代理呼叫会话控制功能（Proxy Call Session Control Function，P - CSCF）。

1）NFV 云分布在运营商的网络中，并且它可以由设计用于零接触、自动化、大规模部署在中心局[⊖]和数据中心的元件构建。

2）NFV 云可以利用网络服务并与网络服务集成，以便为服务提供完整的端到端保证。

3）NFV 云是开放的，因为它必须能够促进来自不同供应商并使用不同技术的应用程序正常运转。

4）NFV 云通过自动化和统一服务提供商可能拥有的多种服务，实现一种全新的操作模式，如分布式云位置和应用程序生命周期（将在第 7 章做进一步介绍）。NFV 云必须能够提供高度的安全性。关于这个问题，请参阅 TMCnet 发布的白皮书，其中概述了作者对这一主题的看法。

毫无疑问，这项最新的前沿技术向我们展示了目前云已经足够成熟，可以变革更多的传统行业，如能源行业。未来几年，我们将看到云对这些行业的财务业绩和竞争力所带来的根本影响。

参考文献

[1] Weinman, J. (2012) The Business Value of Cloud Computing. John Wiley & Sons, Inc, New York.

[2] Chou, T. (2010) Cloud: Seven Clear Business Models, 2nd edn. Active Book Press, Madison, WI.

[3] Hofmann, M. and Beaumont, L. R. (2005) Content Networking: Architecture, Protocols, and Practice (part of the Morgan Kaufmann Series in Networking). Morgan Kaufmann/Elsevier, Amsterdam.

[4] Camarillo, G. and García - Martín, M. - A. (2008) The 3G IP Multimedia Subsystem (IMS): Merging the Internet and the Cellular Worlds, 3rd edn. John Wiley & Sons, Inc, New York.

[5] Olsson, M., Sultana, S., Rommer, S., et al. (2012) EPC and 4G Packet Networks: Driving the Mobile Broadband Revolution, 2nd edn. Academic Press/Elsevier, Amsterdam.

⊖ 中心局（Central Office）是承载一个或多个交换机的电信设备的建筑物。

第3章 CPU虚拟化

本章将介绍虚拟机的概念以及实现它的技术。该技术相当复杂，因为它涵盖了计算机架构、操作系统甚至数据通信的发展。这些问题在这里对云计算来说至关重要，因此我们将着重对其进行介绍。

为此，本章的名称不是特别的准确：本章的内容不仅是 CPU 的虚拟化，而是整个计算机，包括它的内存和设备。鉴于此，如果不是在虚拟化的概念中，处理 CPU 的部分是最重要和最复杂的话，省略 CPU 一词可能会更为准确一些。

从最初的动机和一些历史中，可以追溯到 20 世纪 70 年代初期，以计算机架构的基础知识为切入点，来了解程序控制的具体内容和实现方式。之所以需要着重介绍这个问题，是因为它是安全性问题的核心：通过操纵程序控制可以影响主要的安全攻击。

在解决架构和程序控制之后，将有选择地总结操作系统中最相关的概念和发展。幸运的是，在这一问题方面有很多优秀的参考书，使我们得以进行深入的研究，专攻虚拟化中的关键问题和难题（实现虚拟化的实体，即虚拟机管理程序，实际上是一个“运行”常规操作系统的操作系统）。将解释有关进程的重要概念并列出一些重要的操作系统服务。还将讨论虚拟内存的概念，并展示它是如何实现的——这是一个有趣的发展过程，同时为介绍更为广泛的虚拟化方面的内容做好准备。

一旦完成了这项准备工作，本章将最终阐明虚拟机的概念。将主要关注于虚拟机管理程序，它们的服务、内部运作和安全性问题，所有的这些都通过实例进行说明。

3.1 动机与历史

回到 20 世纪 60 年代，随着计算机的发展越来越快，使用它们的机构和企业在决定是否要更换旧系统时，往往会权衡利弊。其中主要的问题与现在是一样的：软件成本的改变，特别是因为在当时这些成本比现在高得多，可预测性更差。假设一家企业有 3 台或者 4 台老旧的计算机，其中安装了所有的程序，并已经将维护程序设置到位，如果要将软件迁移到一台新的计算机——即使这台计算机比所有传统的机器加起来还要快——企业仍然需要充分地权衡利弊，因为最重要的还是经济问题，如图 3.1 所示。

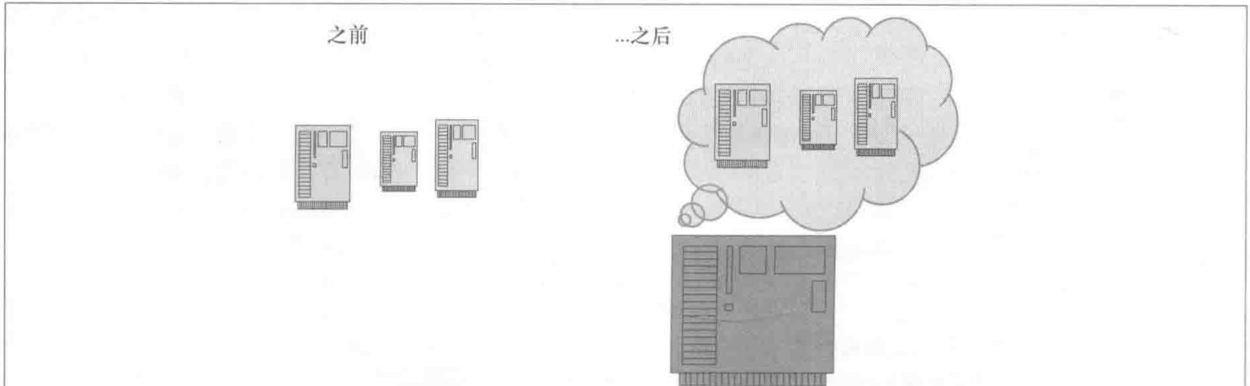


图 3.1 虚拟化前后的计算环境

但是，随着业务的增长，他们的计算需求也随着不断地增长。业内正在努力解决这个问题，在这

方面 IBM 和麻省理工学院主导着一些相关的研究。起初在 20 世纪 60 年代，在 IBM 的 System 360 Model 67 中已经实现了时间共享（即并行运行多个应用程序进程）和虚拟内存（即为每个进程提供独立的全地址范围连续的内存阵列），但是这些技术不足以将多台“整机”移植到一台机器上。换句话说，让独立机器的操作系统以一个单独的用户进程的形式运行在新的机器上，这种解决方案其实并不简单，其原因将在本章后面内容中进行详细的研究。简而言之，主要障碍是（并且仍然是）操作系统的代码需要使用的部分指令的权限，对用户程序来说是不可用的。

克服这一障碍的唯一方法是开发一种超级操作系统来监督其他操作系统的运行。因此，也就产生了“虚拟机管理程序”这一术语。通过 IBM 和麻省理工学院在剑桥科学研究中心的联合研究，最终推出了“控制程序/剑桥监控系统（Control Program/Cambridge Monitor System，CP/CMS）”。该系统经历了 4 个主要的版本，成为最终实现虚拟机管理程序的 IBM VM/370 操作系统的基础。CP/CMS 的另一个具有开创性的成果是创建了一个用户社区，这个用户社区预示了今天的开源运动。CP/CMS 代码被免费提供给 IBM 的用户。

IBM VM/370 于 1972 年对外公布。有关它的介绍和历史在 Robert Creasy 的著名论文[1]中有很好的介绍。CMS，后来更名为会话监控系统，作为其中的一部分。这是一个巨大的成功，不仅因为它实现了将多个系统移植到一台机器上的原始目标，而且还因为它有效地启动了虚拟化行业，这是云计算的一个决定性的推动者。

从那时起，为小型机和以后的微型计算机所开发的所有硬件都部分或全部满足了虚拟化的需求。类似地，软件的开发也解决了与硬件开发相同的需求。

接下来，将分析虚拟化技术方面的问题，同时将它的主要成果总结如下：

1）节省成本（在空间、人员和能源方面——注意绿色环保），用一台机器的运行代替多台物理机器的运行。

2）灵活地投入使用（否则浪费）计算能力。

3）几乎可以立刻实现克隆服务器（例如，为了调试的目的）。

4）为了特定的目的，隔离软件包（通常是出于安全考虑）——不需要购买新的硬件。

5）能够以较低的成本并且可以立刻实现机器的迁移（例如，当负载增加时）——通过网络甚至记忆棒即可实现机器的迁移。

其中最后一个成果（即将虚拟机从一台物理机器迁移到另一台物理机器的功能）被称为实时迁移。在某种程度上，它的目的与将虚拟化具体实现（即将多台机器整合到一台物理主机上）的目的完全相反。实时迁移需要被用来支持弹性化，因为将机器转移到具有更多内存和减少负载的新主机上，可以提高它的性能特征。

3.2　计算机体系结构入门知识

本节内容可以让本书的体系更加完备，内容更加完整。它对于我们理解最重要的虚拟化问题至关重要，特别是在安全性问题方面。熟悉计算机体系结构的读者可以自行跳过这部分内容。在这部分的内容中，更重要的是介绍了主要的编程控制结构（程序调用、终端和异常处理）。对于希望了解更多内容的读者，推荐一本名为“Workhorse of Computer Science Education”的教材[2]㊀，其中介绍的内容更为丰富。

3.2.1　CPU、内存和 I/O ★★★

图 3.2 描述了几乎所有需要了解的计算机的构建模块。对于这些模块，本书将逐步深入地展开介绍。

计算机的三大组成部分包括：

1）中央处理单元（Central Processing Unit，CPU），负责实际执行程序。

2）计算机内存［技术上称为随机访问存储器（Random Access Memory，RAM）］，其中存储着程序

㊀　原书参考文献［2］与此处不一致。——译者注

和数据。

3）输入/输出（Input/Output，I/O）设备，例如显示器、键盘、网卡或磁盘。

所有这3个部分都通过“总线”相互连接，这也使计算机可以进行扩展，从而包含更多的设备。

RAM中的Random一词（与“顺序”的意思相反）意味着存储器被作为数组，通过存储器位置的索引进行访问。该索引被称为内存地址。

需要注意的是，磁盘实际上也是一种类型的存储器，只是读取速度比RAM慢得多。另外，与RAM不同，磁盘和其他永久存储设备上的存储器是持久的：即使电源关闭，存储的数据也仍然存在。

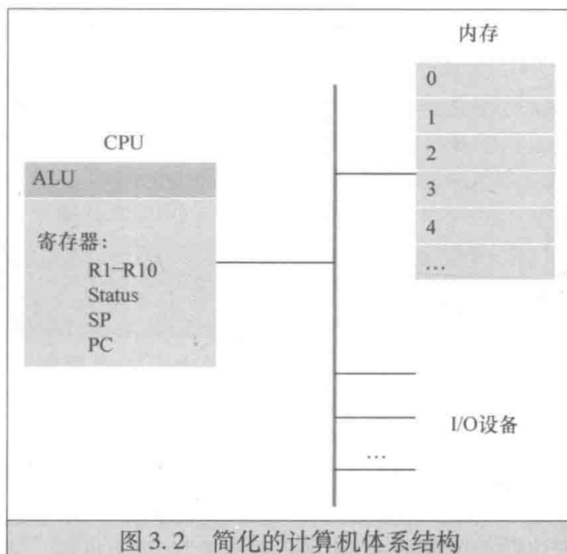


图3.2　简化的计算机体系结构

在存储器领域的另一端，还有比RAM读取速度快得更多的，位于CPU内部。所有的这些存储器都是不同的，这类存储器被称为寄存器。只有寄存器可以执行操作（例如加法或乘法等数学运算，以及一系列的按位逻辑运算）。这是通过将寄存器与算术和逻辑单元（Arithmetic and Logic Unit，ALU）连接的电路来实现的。例如，为了对存储在内存中的两个数字执行算术运算，典型的操作模式是，先将数字传送到寄存器，然后在CPU内执行操作。

一些寄存器（将其表示为R1、R2等）是通用的，而其他的一些寄存器只能用于特殊的需求。为了更好地进行讨论，选取了后一类型中的3个寄存器，它们存在于所有的CPU中：

1）程序计数器（Program Counter，PC）寄存器，总是指向存储下一个程序指令的内存位置。

2）堆栈指针（Stack Pointer，SP）寄存器，总是指向一个进程堆栈的位置。稍后将介绍这个概念。

3）STATUS寄存器，保持执行控制状态。在很多其他的操作中，它存储着与上一次操作结果相关的信息。例如，当一个算术运算结果为零时，就会将STATUS寄存器中被称为零比特位的标志置位，类似地，在STATUS寄存器中还有正比特位和负比特位标志。所有这些都用于转移指令：JZ为零跳转；JP为正跳转；JN为负跳转。反过来，这些指令被高级语言用来实现条件if语句。对于虚拟化来说，另一个非常关键的是对STATUS寄存器的使用（稍后讨论），用于指示CPU必须在跟踪模式下工作，即每次执行一条指令。需要的时候，将介绍一些新的标志。

总的来说，所有寄存器值的集合（有时称为上下文）构成了CPU所执行的程序状态。执行中的程序被称为进程[㊀]，这是一个非常模糊的定义，这里通过一个比喻来阐述它的用途。可以把一个程序看作一本食谱，将CPU看作使用厨房用具的厨师，然后可以把进程定义为烹饪食谱中所描述的具体菜肴的动作。

当厨师切换到准备另一道菜肴时，只要菜肴的状态被记住（即食谱中的具体步骤），厨师就可以同时烹饪多个菜肴。例如，厨师可以将烤肉放入烤箱，设置定时闹钟，然后开始制作点心。当闹钟响起时，厨师将暂时放弃甜点并处理烤肉。这样做，厨师必须知道是要继续烘烤，还是把它从烤箱里拿出来。一旦烤肉处理完毕，厨师就可以恢复在甜点上的工作。但是厨师还需要记住甜点制作工作被搁置的地方。

由现代操作系统维护的多编程做法，是将CPU的状态存储在进程堆栈上，这就带来了CPU内部运行的问题，供我们进行研究。

为了完成本节内容（并对图3.2进行相对简单的介绍），此处将对现代CPU的内部运行问题进行一个基本的了解，如今的CPU内部可能有不只一组相同的寄存器。至少为用户模式保留一个寄存器集——应用程序在其中执行，其他的寄存器集供系统（或监控或内核）模式使用——只有操作系统的软件能够在其中执行，其原因将在后面进行介绍。

㊀ 在某些操作系统中，“线程”实际上表示程序的执行，而将“进程”保留用来表示共享相同内存空间的一组线程。但是，这里有意不区分这两者之间的区别。

3.2.2 CPU 的工作原理 ★★★

CPU 的概念相当简单，但需要注意的是，CPU 本身不会“理解”任何程序。它只能处理用它自己的、CPU 特定的机器代码编写的单条指令。因此，它只能保持这条指令的处理状态。一旦这条指令执行完毕，CPU 就会“忘记”它所做的一起，并开始进入到下一条指令执行的新周期。

虽然不需要了解特定 CPU 所有的机器代码指令，但是为了理解它的工作原理，必须要掌握其中的一些基本概念。

正如 Donald Knuth 在他开创性的研究[3]中所提到的，“对计算机较感兴趣的人应当学习机器语言，因为它是计算机的基本组成部分”。如今这句话变得更重要了——没有对机器语言结构的理解，也就无法掌握虚拟化。

幸运的是，它所涉及的问题是非常简单的，可以通过几个指令来解释这些问题。为了简单起见，本书将避免参考现有 CPU 的指令，而只是以自己的指令为例。最后，CPU 将指令“视为”比特串，最终构成机器级代码，不过不需要考虑这个级别上的问题。只需要了解用来代表这些指令的文本“汇编语言”即可。

每个指令都由其操作码 opcode 组成，它指定了要执行的操作，后面跟着操作数列表。首先，要想对内存中存储的变量进行操作，CPU 就必须先将该变量加载到寄存器中。

举一个简单的例子：要想把分别存储在地址 10002 和 10010 的两个数相加，程序就必须先将这些数传送到两个 CPU 寄存器中，如 R1 和 R2。这是通过 LOAD 指令来实现的，它只完成这样的工作：将目标加载到寄存器中。产生的程序如下：

```
LOAD R1 @ 10002
LOAD R2 @ 10010
ADD R1, R2
```

其中符合汇编语言约定的字符“@”表示间接寻址。换句话说，“@”后面的数字字符串表示一个地址，从这个数字值指示的地址内将变量的值加载到寄存器中，而不是加载这个数字值本身。如果想表示寻址是立即的，也就是说，要加载数字字符串的实际值——使用字符“#”来处理它，如`LOAD R1, #3`。

在上面的小程序中，最后一条指令实现两个寄存器的值相加并将结果存储在第二个操作数寄存器 R2 中（正如机器语言所定义的那样）。

在大多数情况下，程序需要将结果存储在其他地方。STORE 指令用于完成这项工作，它实际上是 LOAD 指令的相反过程。假设变量 x、y 和 z 分别位于地址 10002、10010 和 10020，可以用指令`STORE R2, @ 10020`来执行 C 语言的赋值语句：`z = x + y`。

类似地，除了算术指令 ADD 外，其他的指令几乎不需要进行任何额外的说明。这里需要简单提一下的是逻辑指令：AND 和 OR，它们是按位执行相应的操作的。因此，指令`OR R1, X`是用 X 中值为 1 的比特位对 R1 中的相应比特位置位；而指令`AND R1, X`则是用 X 中值为 0 的比特位对 R1 中的相应比特位复位。这些逻辑指令再加上 SHIFT（根据参数值将寄存器比特位向右或向左移位指定位数，同时复位被移位的位）指令可以实现所有位模式的操作。接下来，将介绍一些其他的指令。现在准备先通过一个非常简化的描述来看一下 CPU 的工作机制，如图 3.3 所示。后面，将继续介绍与这一描述相关的重要细节。

图 3.3 简化的 CPU 循环（第一近似值）

CPU 像时钟一样工作，这与机器世界非常类似。计算机的所有操作都是按照被称为计算机时钟的设备所发射的脉冲频率来进行的，正如部分机械时钟是根据摆锤摆动而工作的一样。为此，CPU 的速度可以通过它能够支持的时钟频率来测量。

所有 CPU 都执行严格的无限循环，在这个过程中，不断地从内存中获取指令并执行指令。一旦完成，一切将重新开始。除了寄存器中剩余的内容外，CPU 不携带上一条指令的内存。

如果把小程序放在内存位置 200000 中㊀，那么必须用这个值加载 PC 寄存器，以便 CPU 开始执行这个程序的第一条指令。然后，CPU 将 PC 步进到下一条指令，恰好在地址 200020 处。因此，可以很容易地看到程序的其余部分如何执行。

不过在这里，对于程序的每个指令来说，下一个指令碰巧就是内存中的下一条指令。一般编程绝对不是这样，因此 CPU 需要更为复杂的控制转移能力，接下来将讨论这一问题。

3.2.3 程序内控制转移：跳转和过程调用 ★★★

至少，为了执行 if – then – else 逻辑，需要一个指令，强制 CPU “跳转” 到内存中与上一条指令地址不连续的地址上的指令，JUMP 就是一条这样的指令。它唯一的操作数是一个内存地址，它的执行结果是将该地址变成 PC 寄存器的值。

该系列指令中的另一个指令是 JNZ（如果非零即跳转），对该指令唯一的操作数所提供的地址执行条件转移。这里的非零指的是 STATUS 寄存器的 0 比特位的取值。每次算术运算结果或逻辑运算结果为零时，CPU 都将通过 ALU 电路进行内部处理。执行该指令时，CPU 只会将 PC 的值更改为操作数的值。STATUS 寄存器通常保存其他条件位，以指示数字结果是正还是负。为了使程序员的工作更加容易（并且其结果更快），许多 CPU 提供了条件转移指令的其他变体。

对所有现代 CPU 而言，更有趣也是最基本的是将控制转移给其他过程（一段程序）的指令。我们将这种指令称为 JPR（跳转到程序）。这里，CPU 通过将 PC 当前值（根据图 3.3，该值最初指向内存中的下一条指令）自动存储在 SP 指向的堆栈㊁上来帮助程序员。这样，SP 的值就被适当地改变，从而使 CPU 能够将控制权返回到程序中调用过程的位置。为了实现这一点，有一个无操作数的指令 RTP（从过程中返回），它可以将之前的值弹出堆栈并恢复 PC 的值。不过，这条指令必须是每个过程体中的最后一条指令。

这里有几个重要的问题需要考虑。

首先，我们看到只有使用 JUMP 指令才能实现一个类似的结果。毕竟，程序员（或编译器）可以添加几个指令来将 PC 存储在堆栈上。从一个 JUMP 到该过程结束时弹出的 PC 值，该任务就完成了。因此，即使 CPU 没有堆栈的概念，一切也都可以正常工作，所以它可以是一个用户定义的结构。不过，现代 CPU 是按照这里描述的方式进行设计开发的，其中的两个主要原因在于：①使过程调用更快（通过避免获取其他额外的指令）；②保持良好的编码习惯，使 ALGOL 语言及其衍生物的适应性变得简单直观（一种语言导向型设计）。正如已经注意到的，递归就是用这种技术构建的。

其次，进程作为程序执行的概念现在应该更加清晰。实际上，堆栈跟踪着当前代码主线之外的控制转移。有趣的是，在 20 世纪 80 年代，Borroughs 公司的程序人员高度创新，在当时 CPU 架构还是 ALGOL 导向的时候就交替地使用了进程和堆栈这两个词。这是理解进程的一个非常好的方式——因为其堆栈可以有效地表示其自己，而堆栈始终跟踪单个线程的执行。

第三，这种结构开始为多重处理的支持机制揭开序幕。假设 CPU 可以将其所有的状态存储在进程堆栈中，然后恢复它们（这种功能将在下一节中进行介绍）。可以想象，一个 CPU 可以通过在各个堆栈之间切换来同时执行不同的进程。

第四，这是一个重大的安全问题。事实上，当从一个程序过程返回时，CPU 会弹出堆栈并将其视为 PC 值，不管这个值之前存储在哪里。这就意味着如果有人使用另一个内存地址替换 PC 的原始存储值，那么 CPU 将自动开始执行该内存地址处的代码。这种情况已经被用于传播计算机蠕虫病毒。典型的 PC 重写技术是将缓冲区作为程序过程的参数（因此出现在堆栈上）。例如，如果缓冲区被读取的用户字符串填充满，并且程序代码没有检查缓冲区的限制，则可以通过将该字符串进行精心的构造来传递：蠕虫代码和指向该代码的指针，从而用该指针重写 PC 的存储值。1988 年原始的 Morris 蠕虫已经成功尝试了这一技术（详见本章参考文献 [4]，该文献从蠕虫的发展历史展开，对这一技术进行了全面

㊀ 在这一点上，故意忽略定义表达地址的内存单位（即字节或字）。

㊁ 堆栈一般设置在较高的存储器地址中，从而使其向下“增长”：将 n 个存储单元放在堆栈上，将导致 SP 的值减少 n。



彻底的说明)[1]。对此，将在本章的最后一节讨论相关的安全性问题。

对于以下内容，重点在于对如何使用堆栈来实现过程调用进行详细的说明。

有了 CPU 的少量帮助，剩下的工作则交给程序员（如果程序是程序员使用汇编语言编写的）或编译器（如果程序是程序员使用高级语言编写的）来完成对过程参数的处理。长期以来的做法是，在调用过程之前将它们放在堆栈上。

在过程调用期间，另一个必须要程序员（或编译器）解决的问题是对过程局部变量的管理。同样，在这方面长期以来的做法是在堆栈上分配所有的局部内存。这样做的最大的好处是可以启用递归：每当一个程序被调用时，它的参数和局部内存都能够与之前被调用的程序区分开。

图 3.4 通过跟踪执行示例程序[2]以及执行过程中每条指令对应的进程堆栈状态，来说明了这一点。这里，一个存储在位置 1000000 的过程被主程序所调用。该过程有两个参数，分别存储在位置 20002 和 20010 中。

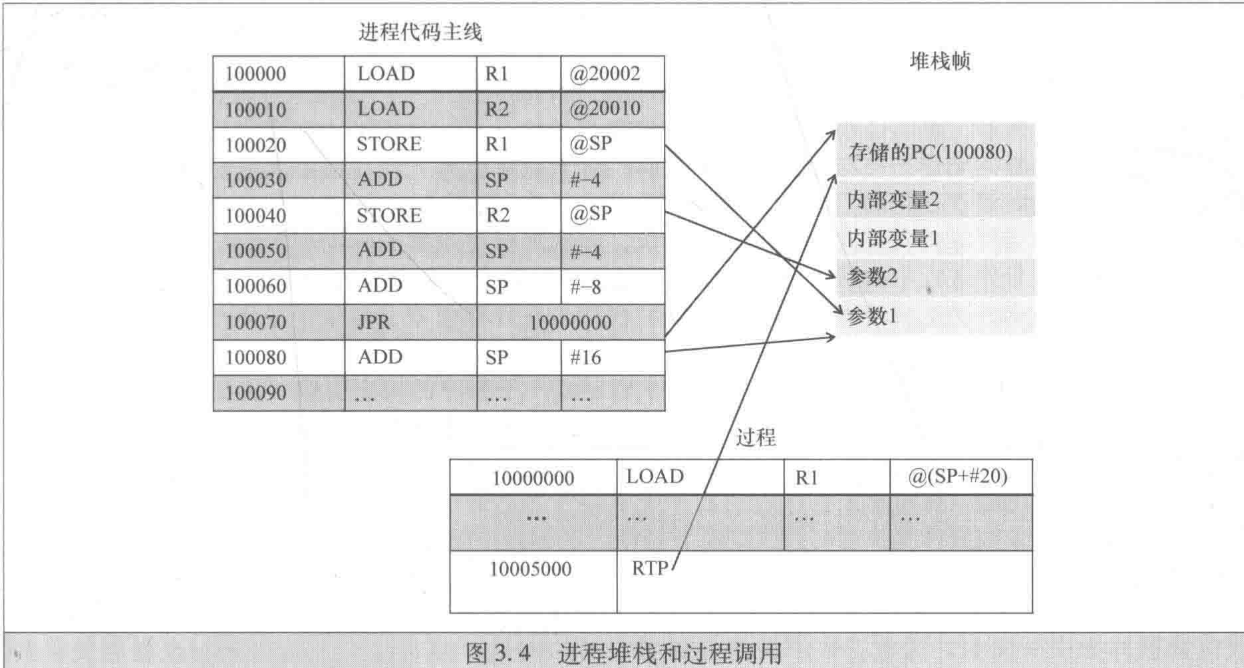


图 3.4 进程堆栈和过程调用

前六个指令实现了在堆栈上推送过程参数的动作。需要注意的是，在这里我们认为每个参数为 4 个单位长，因此使用`ADD SP #-4`；当然，按照惯例，当堆栈减少时，堆栈指针的值也随之减少。

第七条指令（位于地址 100060）用来在堆栈上为过程准备内部存储空间，在这个例子中，需要为这两个 4 个单位长的变量准备 8 个单位的存储空间。

在第八条指令中，实现过程代码的调用。这一次，CPU 将 PC 上的值（也是 4 个字节长）推送到堆栈上；然后，CPU 将该过程的地址加载到 PC 中。此时，该过程的堆栈帧已经建立。

执行该过程的第一条指令是获取第一个参数的值，我们知道这个值位于堆栈上，恰好高于堆栈指针 20 个单位。类似地，相对于堆栈指针的值，间接访问另一个参数和内部变量。（我们故意没有显示堆栈的实际内存位置：使用这种寻址方式，只要堆栈正确初始化，该位置就是无关紧要的！这个功能非常强大，它消除了绝对寻址的需要。同样，该功能可以直接支持递归，因为相同的代码将使用一组新的参数和新的内部存储空间执行程序）。

过程完成后，执行 RTP 指令，使 CPU 弹出堆栈并将程序计数器恢复为存储值。因此，程序控制返回到主程序。示例中的最后一条指令将堆栈恢复到原始状态。

这里需要注意的一个小问题是，过程可能是一个函数，也就是说，它可以返回一个值。该值如何传递给调用者呢？将其存储在堆栈上是实现这一目的的一种方法。不过，C 语言编译器已经通过寄存

[1] 那时，它利用了 UNIX Finger 实用程序的漏洞。令人惊讶的是，同样的漏洞仍然是有意义的。

[2] 该示例程序故意未进行优化。

器实现返回值的传递，因为它的速度更快。

以上讨论了有关过程调用的相关内容。由于新的控制转移形式的出现，接下来将进入到 CPU 机制的下一个细节层面。

3.2.4 中断和异常——CPU 循环细节★★★

到目前为止，前面设计的简单 CPU 只能处理一个进程（本节将继续使用这一限定）。该进程的行为是由其主线代码中的指令集以及将“控制转移”到的过程所确定的，但仍然是绝对可预测（有时称为“确定性”或“同步的”）的方式。这种类型的 CPU 存在于头几十年的计算机中，并且大体上可以被用来执行内存中的运算处理。特别适合于执行复杂的数学算法，只要不需进行太多的 I/O 设备访问。

但是，当需要处理 I/O 请求时会发生什么呢？使用目前的设计，唯一的解决方案是拥有一个知道如何访问给定 I/O 设备的子程序，如一个磁盘或一个打印机。可以假设这样的设备是内存映射的：在主内存中有一个写这种命令（读或写）的位置，将一个指针传递给这类数据驻留（或将被读入）的数据缓冲区，以及可以检查操作状态的位置。启动命令后，进程代码只能执行紧凑循环检查该状态，除此之外不能执行任何其他的操作。

从历史的角度来看，事实证明，对于 CPU 密集型的数值计算，这样的方案多少还能令人满意，因为 I/O 处理不是频繁的动作——与计算相比。然而，对于商业计算来说，需要大量使用磁盘、多个磁带机和打印机，因此浪费在轮询上的 CPU 周期是主要的性能瓶颈。随着设备越来越复杂，这个问题将进一步恶化，因为需要保持交互式设备专门的协议，而这需要更频繁的轮询和等待。例如，在需要将打印机的每个字节分别进行单独写入的情况下，根据打印机对上一个字节的响应处理执行具体的操作。出于这个原因，轮询的功能被转移到 CPU 循环中，代价是在计算模型上发生了变化，而且在这方面也发生了戏剧性的变化。

变化的要点是：鉴于之前是由程序员在程序中确定调用子程序的特定位置（无论是从主线或其他子程序），现在可由 CPU 根据与设备的通信自行决定对某个子程序的调用。这样，CPU 可以有效地中断程序中的指令链，这意味着 CPU 还需要能够准确地回到这条指令链中被中断的位置。这里涉及的术语很多：由设备发出的，相对于正在执行的程序能够异步到达的信号，被称为中断；从中断的那一刻开始，直到原始指令线程恢复执行时产生的所有操作，被称为中断处理。

当然，用于处理来自设备输入的实际代码——中断处理程序——必须由程序员编写。但是，由于这个例程不会被其他程序显示调用，所以它必须被放置在 CPU 可以找到它的指定内存位置，该位置被称为中断向量。通常，每个设备都拥有自己的向量，或者甚至一组用于与设备相关联的不同事件的向量。实际上，这方面的内容是非常复杂的，但是此处的介绍不需要达到这样的详细程度。在初始化时，程序必须将正确的中断程序地址放入分配给中断向量的插槽中。当 CPU 检测到来自设备的信号时，它将跳转到相应的中断程序。当中断被服务后，控制将被返回到中断执行点，恢复原始程序的执行。

该机制提供了处理与执行程序异步的外部事件的方法。由于一些原因（稍后将介绍清楚），同样的机制也被用于处理实际上与执行程序同步的某些事件。它们是同步的，因为它们正是由所执行的指令引起的（因此，作为结果，最终可能无法执行）。此类事件的重要示例如下：

1）计算异常（如尝试除以零）。

2）内存引用异常（例如尝试读取或写入内存中不存在的位置）。

3）尝试执行不存在的指令（或在当前上下文中非法的指令）。

4）直接插入处理异常的显示请求（称为陷阱）。在 CPU 中，这是由一条指令（形式为 TRAP <trap number>）引起的，该指令可以用这个陷阱把一个参数“陷阱号”与一个特定的中断向量进行关联，以便不同的陷阱号可以被不同地处理。

有关陷阱指令基本需求的内容将在后面详细说明，一个应用就是在调试时用来设置断点。当开发人员希望程序在特定的位置停止，以便可以检查内存时，结果指令将被陷阱指令替换，如图 3.5 所示。

虚拟机管理程序也是用这一技术来处理非虚拟化指令的，稍后会详细进行说明。

图 3.6 对以下讨论的内容进行了说明。



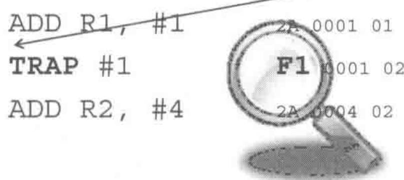


图 3.5 设置断点

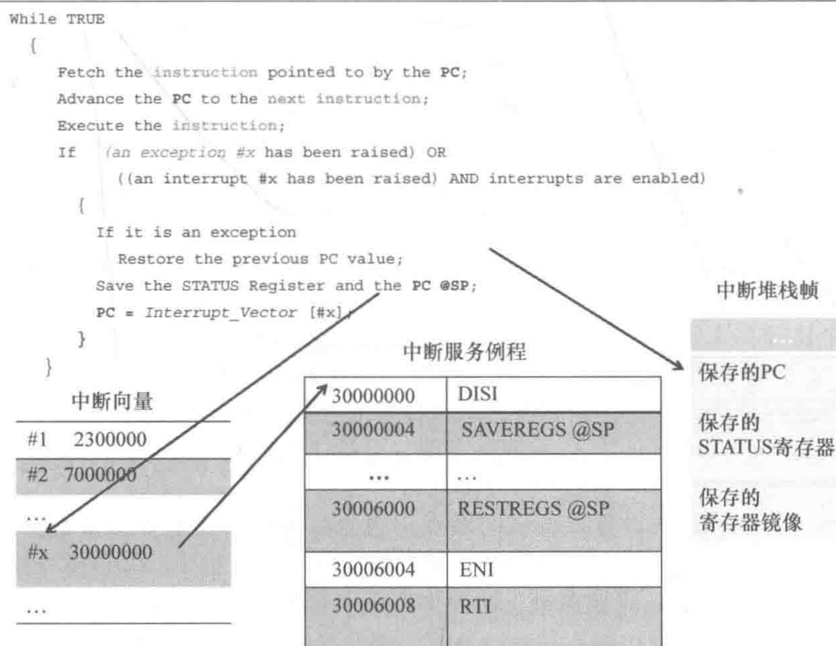


图 3.6 另一种类似 CPU 循环的过程

我们准备对图 3.3 中简化的 CPU 循环进行修改，新的循环会检查中断或异常信号。需要注意的是，就 CPU 而言，处理是相当确定的：在当前指令执行结束时进行检查，无论信号何时到达。CPU 的内部电路可以让它确定中断或异常的类型，这体现在中断号 x 中。该数字作为中断向量表的索引[⊖]。在处理中断和处理异常之间存在着重大的区别：导致异常的指令尚未执行；因此，要被存储的 PC 的值必须保持不变。除此以外，中断和异常之间的处理不存在区别，并且为了避免冗余重复，本节的其余部分也将使用“中断”一词来表示“中断或异常”。

图 3.6 左边的表表明，处理中断 x 的中断服务例程的开始位置在 30000000。CPU 处理这一代码的过程与程序处理该代码的过程相同。不过，在特殊的情况下，还需要有一套特殊的操作过程。不同的 CPU 在这里做不同的事情，我们的 CPU 多少能够做一些现代 CPU 做的事情。

为此，我们的 CPU 通过将当前 STATUS 寄存器的值保存在进程堆栈中来启动程序。原因在于，执

⊖ 在早期的 CPU 中，中断向量表通常处于低内存（存储器）地址中，一般从地址 0 开始。现代 CPU 有一个指向该表的特殊寄存器，可以将其放置在存储器的任何位置上。从这个意义上来说，即使在一个多用户环境中仍然只处理单用户/单进程的执行，也可以使用这种特殊的寄存器。通过这样的寄存器设计，有助于在多用户环境下实现对内存特定部分访问保护的目的，该目的是操作系统和虚拟机管理程序的一项重要任务。在下一个 CPU 循环修改中，将解决这个问题。

行新指令时，根据上一条指令的结果，STATUS 寄存器中标志位体现的所有条件都将消失。例如，如果程序在某个数字大于另一个数字时需要进行分支，则可能会导致以下几个指令的产生：

1）两个指令将相应的值加载到 R0 和 R1 中。

2）一个从 R0 中减去 R1 的指令。

3）分支的最后一条指令，取决于减法的结果是否为正。

最后一条指令的执行取决于根据第三条指令执行结果而设置的标志，因此如果进程在这条指令之后被中断，那么当它继续，必须保留该标志。

STATUS 寄存器保存后，CPU 将保存 PC 的值，就像使用过程调用堆栈帧执行的操作一样，然后开始执行中断服务例程。

后者的第一条指令必须是 DISI（禁用中断）执行。实际上，从 CPU 发现一个正在等候的中断，直到返回到这条指令，一切都是由 CPU 本身完成的，它不会中断它自己。但是来自同一设备的下一个中断可能已经到了。如果 CPU 处理它，这个中断例程将被中断。有问题的设备（或恶意操纵）将导致一组递归调用，造成堆栈增长，直到它溢出，最终这将导致整个系统关闭。因此，在执行一些关键指令期间，至少要在很短的时间内，有必要将中断禁用掉。

接下来，我们的示例中断服务例程使用 SAVEREGS 指令将其余的寄存器（不包括 PC 和 STATUS 寄存器）保存在堆栈中㊀。这样，实际上整个进程的状态得到了保存。即使所有执行的中断服务例程都使用该进程堆栈，它也可以独立于进程本身执行，或与进程同时执行，而完全不必了解具体发生了什么。

中断服务例程的其余代码处理其他需要做的事情，当它完成后，它会恢复进程的寄存器（通过 RESTREGS 指令），并允许中断（通过倒数第二条指令 ENI）。最后一条指令 RTI，告诉 CPU 恢复 PC 和 STATUS 寄存器的值。下一条 CPU 将执行的指令正是进程被中断时的那条指令。

通常在 CPU 的情况下，还有一个指定的 STATUS 寄存器标志（位），指示中断是否被禁止。DISI 只设置此标志，ENI 将其复位。

调试工具是一个比较有说明性的例子，如前所述：该工具可以让我们设置断点，以便在到达断点时分析计算的状态㊁。目的是让用户在代码的特定指令处设置断点，使得当程序执行到该指令时停止，通过输入命令交互地进行分析。此时，调试器显示寄存器的相关信息并等待用户的下一个命令。

我们的调试工具由 command_line（ ）子例程实现，可被 TRAP #1 服务例程调用，能够接受以下 6 个命令：

1）Set <location>，在代码中设置要调试的断点。因为这可能需要重置，命令的效果是：①这条指令（存储在 <location>）和 <location> 的值都存储在各自的全局变量中。②该指令被替换为 TRAP #1，图 3.5 描述了这条命令的效果。遵循惯例，使用十六进制符号来显示内存值。因此，TRAP #1 指令的操作码恰好是 F1。

2）Reset，将被陷阱替代的原始指令返回到它的位置。

3）Register <name>，<value>，用它的值设置了一个命名的寄存器。所有 CPU 寄存器（除了 PC 和 STATUS 寄存器需要单独的处理）的镜像都被保持在一个全局结构 registers_struct 中，所以当用户输入一条命令时，如 Register R1，20，将会执行一次赋值“registers_struct. R1 = 20;”。

4）Go，基于 registers_struct 中存储的寄存器的各个值，开始执行代码。

5）Show <memory_location>，<number_of_units>，简单地提供了一种由参数指定的内存块的核心转储功能。

6）Change <memory_location>，<value>，这可以让我们更改内存单元的值。

Go（ ）过程和中断向量例程 TRAP_1_Service_Routine（ ）如图 3.7 所示。稍后将对这两个程序进行介绍，先从初始化开始。起初，我们要：

1）将指向 TRAP_1_Service_Routine（ ）的指针地址存储在 TRAP #1 指令的中断向量中（该位置与具体的 CPU 有关，由 CPU 手册提供）。

㊀ 不同的 CPU 会以不同的方式来处理这个问题。一些 CPU 采用将所有的寄存器自动保存在堆栈上；还有一些 CPU 希望中断例程只保存那些需要保存的目标。

㊁ 在本章后面的内容中，将定义“自省”这个术语。我们要描述的机制是一种简单自省的例子。

2）执行 TRAP #1 指令，这将导致 TRAP_1_Service_Routine（ ）的执行。后者调用 command_line（ ）过程，它会提示用户输入命令，然后解释命令。

Go( )代码

```
Go()
 {
    #DISI; /*     禁用中断     */
    registers_struct.SP =
       registers_struct.SP +4;

   #RESTREGS @registers_struct;
   #STORE PC  @SP+8;
   #STORE STATUS @SP+4
     /*
        将PC和STATUS放置在堆栈
        帧中的正确位置
                              */

   #ENI; /*      激活中断      */
   #RTI;
 }
```

TRAP #1服务例程

TRAP #1向量存储TRAP_1_Service例程的地址

```
#DISI; /*   禁用中断   */
#SAVEREGS @registers_struct;
#ENI;  /*   激活中断   */
display(registers_struct, PC,
 STATUS);
command_line();
```

图 3.7 Go（ ）和中断服务例程

现在开始调试。例如，想让一个程序（它的第一条指令位于内存地址 300000 处）执行到位于地址 350000 处的指令时停止。

输入以下 3 个命令：

```
>Register PC, 300000
>Set 350000
>Go
```

当解释器调用 Go（ ）时，它首先弹出堆栈来填补调试器的 command - line 过程帧（在设计中，不会通过执行 RTP 来从这个调用中返回）。然后，它将存储在 registers_struct 中的寄存器值传送给 CPU。对于 PC 和 STATUS 寄存器，分别重复相同的任务，其值必须在堆栈上修改以构建正确的堆栈帧。最后，执行 RTI 指令。

这将使程序移动。当它到达陷阱指令时，TRAP 处理程序将被调用。因此，将会看到寄存器的值和一个提示。可以查看内存，还可以改变变量的值，替换陷阱指令或设置另一个陷阱。

需要注意的是，Go（ ）过程实际上已经完成了中断处理：RTI 指令不是陷阱服务例程的一部分。可以在内存中有多个程序，并且通过适当地修改寄存器，可能会导致另一个程序通过“返回”运行。

戏剧性的是：可以根据需要同时运行多个进程！

为此，为每个进程分配适当的内存（称为堆）用于堆栈和其余的进程内存，在堆中保存它运行时的数据。同时还建立了适当的堆栈帧。在后者中，PC 的值必须指向进程代码的开始，并且 SP 的值必须指向进程的堆栈（其余的寄存器在这一点上并不重要）。只需要执行图 3.5 中最后的 3 条指令。当 RTI 指令执行时，就可以实现我们的目的㊀。

因此，到目前为止使用本书所描述的 CPU，尽管如此简单，但是已经向虚拟化迈进了第一步，即多重处理。有了这一步，可以让多个进程（可能属于不同的用户）共享 CPU。这是“从外部”的视角来看待多处理。“内部”的视角（从进程的视角）就是进程被赋予了自己的 CPU。

拥有 CPU 的感觉是迈向虚拟化的第一步。但是，如果没有虚拟化内存，那么“虚拟”的理想就不可能实现，这让一个进程“认为”它有自己的完整内存空间，地址从 0 开始。

有关这两个方面（CPU 和内存虚拟化）的内容，将在接下来的两节中讨论。事实证明，增加新功

㊀ 这里 RTI 指令的效果与 Igor Stravinsky 的钢琴曲《Petrushka》[5] 中老魔法师的长笛具有相似的效果：施过魔法后，当被长笛的笛音触动时，木偶活了过来并开始跳舞。

能需要改变体系结构。随着多重处理需求的明确，我们的 CPU 将进一步发展，以实现对它们的支持。

3.2.5 多重处理及其要求——操作系统的需求 ★★★

考虑只有两个进程的情况。为了跟踪它们的进度，需要创建并填充一个数据结构，即一个数组，如图 3.8 所示，其中每个条目包含了：

1）进程寄存器的完整值集（PC 初始指向相应程序代码分段，SP 指向堆栈）。

2）进程的状态告诉我们进程是否①等待某一事件（例如 I/O 的完成），②准备运行，或③实际上已运行。当然，这两个进程中只有一个进程可以处于后一种状态（对于这个问题，使用一个 CPU 时，只有一个进程可以运行，无论有多少个其他进程）。

3）分段和页表指针（参见 3.2.6 节），间接指定了进程堆内存的地址和大小（即为其全局变量分配的内存）。

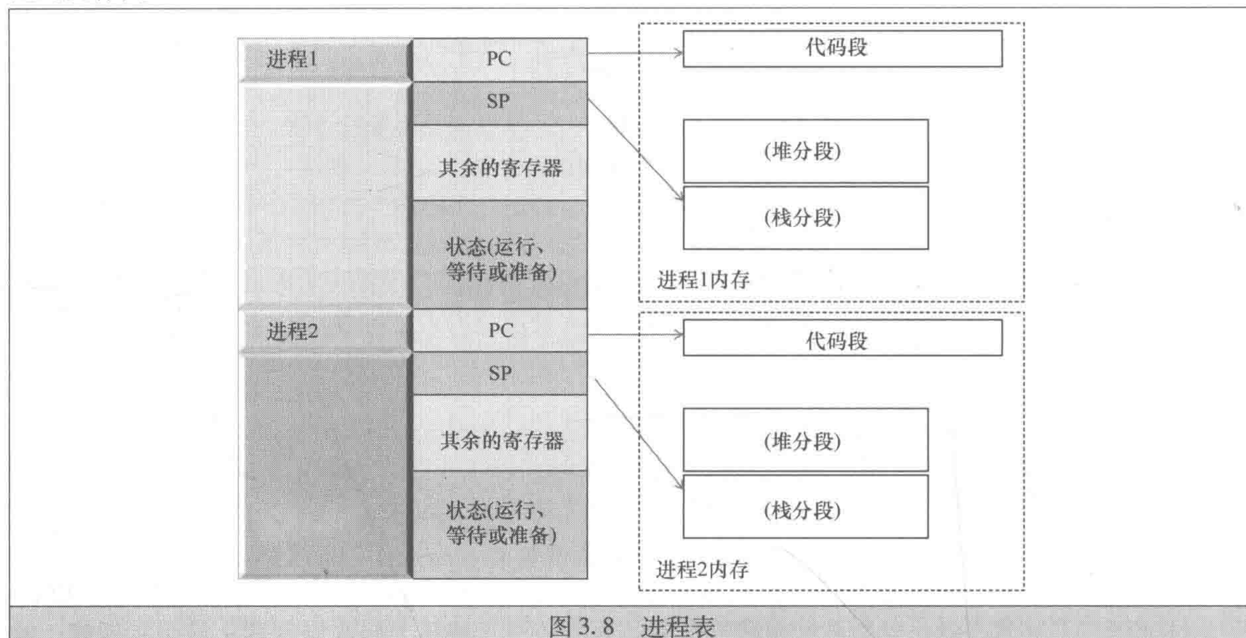


图 3.8 进程表

其他需要内部处理的进程条目也可以容易地想象出来，但是只要使用这种简单的结构（被称为进程表），就可以保持这些进程，用上一节结尾所描述的方法启动它们，并在中断期间介入它们的工作周期。

还可以很容易地看到在这个示例中有“两个”进程，这里所举例的进程数量并不绝对。只要内存和设计允许，该表可以有很多的条目。

需要用来管理进程的程序被称为“操作系统”，其目标为一般的管理提供一个完美的例子：操作系统必须完成很多工作，但无论它做什么都要快速完成，而不会明显地干扰它管理的进程。

这些生命周期中的重大事件发生在中断期间。为此，此时的操作系统只不过是一个库——一组从主线代码或中断服务例程中调用的程序。顺便提一句，系统甚至可以通过处理时钟中断并检查 CPU 绑定进程是否超过了它所占用的时间份额（即允许拥有 CPU 的时间），在这些进程间平均分配 CPU（这个份额被称为量子或时间片。在配置时间对它的值进行赋值，使我们能够支持 CPU 虚拟化声明：在 N 个进程之间共享速度为 θHz 的 CPU[⊖]，将使每个进程被赋予一个速度为 θ/N Hz 的虚拟 CPU）。

总体来说，操作系统负责进程的创建和调度并为它们分配资源。简而言之，这是通过仔细维护一组队列来实现的。在其生命周期中，一个进程总是在改变它的状态。除非进程正在执行，否则它总是在等待着其他事件（例如，文件记录被读取，来自其他进程的消息，或者在它准备执行时等待 CPU）。操作系统确保每个进程始终处于正确的队列中。我们将继续回到操作系统的主题

⊖ CPU 的速度以它所支持的时钟频率来衡量。在这方面，它非常像音乐家演奏使用的节拍器。音乐（指令执行）相同，但可以播放得更快或更慢，这取决于乐曲的节奏和演奏者的能力。

上来，但这个问题需要进行单独的研究。幸运的是，在这部分内容，有很多优秀且全面的参考书[6,7]，对此强烈推荐。

回到所掌握的体系结构上来，需要注意3个基本的问题。

第一个问题是用户程序必须以某种方式了解其代码和数据所在内存中的具体地址，但是这些地址只有在运行时才会知道。关于这个问题，将在下一节中给予解决。

第二个问题是目前没有办法防止一个进程访问另一个进程的内存（无论是出于恶意的原因，还是仅仅简单地由于程序员的失误引起的）。更糟的是，每个进程还可以访问操作系统的数据，因为在这一点上，没有办法将其与用户数据相区分。

第三个问题是无法阻止进程执行某些指令（例如，禁用中断）。如果用户在他的程序中禁用中断，计算机可能会变成聋子和哑巴。顺便提一下，这是很好的也是非常重要的一点，需要一些细化的考虑。因为讨论的系统代码只是一个系统例程库，它需要被一个或另一个进程执行。实际上，进程的程序可以直接调用相应的库程序，也可以在中断期间调用它，这是在中断进程的堆栈中处理的。然而，显而易见的是，只有系统代码才应当被允许禁用中断，而不是用户程序代码。因此，这里的问题还是，用户代码到目前为止仍然无法与系统代码进行区分。

通过认识到CPU必须有两种执行模式，因此最后一个问题已经被解决了。这两种模式是用户模式和系统模式。前者用于用户程序和一些非关键的系统代码。后者用于关键的操作系统代码（或内核）。因此，某些指令必须只能在系统模式下执行。相应地，CPU和现在的操作系统已经发展到支持这一原则。

如何实现它呢？我们观察到系统代码分别在以下情况时会被调用：

1）在用户程序中显式调用。

2）中断（或异常）发生时。

为了确保仅通过指定的门才能进入到系统处理，假设所有的系统调用只能通过“异常处理”的方法来实现，即通过执行TRAP指令才能实现（现在已经履行了前面的承诺，解释了这一指令的进一步使用）。每个需要访问系统资源或外部设备的程序都必须这样做。这将上述第一种情况变为第二种：现在只能通过中断或异常调用系统代码。

进一步假设CPU是在系统模式下启动的，且每当处理中断或异常时，自动进入系统模式。因此，只有当（由内核）明确指示这样做时，CPU才能从系统模式切换到用户模式。

在下文中，主要的想法是确保在用户模式下进程能做的事情是有限的，只能影响进程本身，而对其他进程或操作系统没有影响。为了实现这个目标，首先需要让CPU知道什么样的代码正在运行，是用户代码还是操作系统代码。其次，需要限制在用户模式下执行的动作。再次，该指导原则是CPU在系统模式下启动，并且只有在被明确指示这样做的时候，才能切换到用户模式。当它在用户模式下，CPU的行为必须受到限制。

通过引入一个新的模式标志来让CPU知道操作模式，这是一个指示系统模式（设置时）或用户模式（复位时）的比特位。图3.9描述了管理系统和用户状态之间转换的状态机。

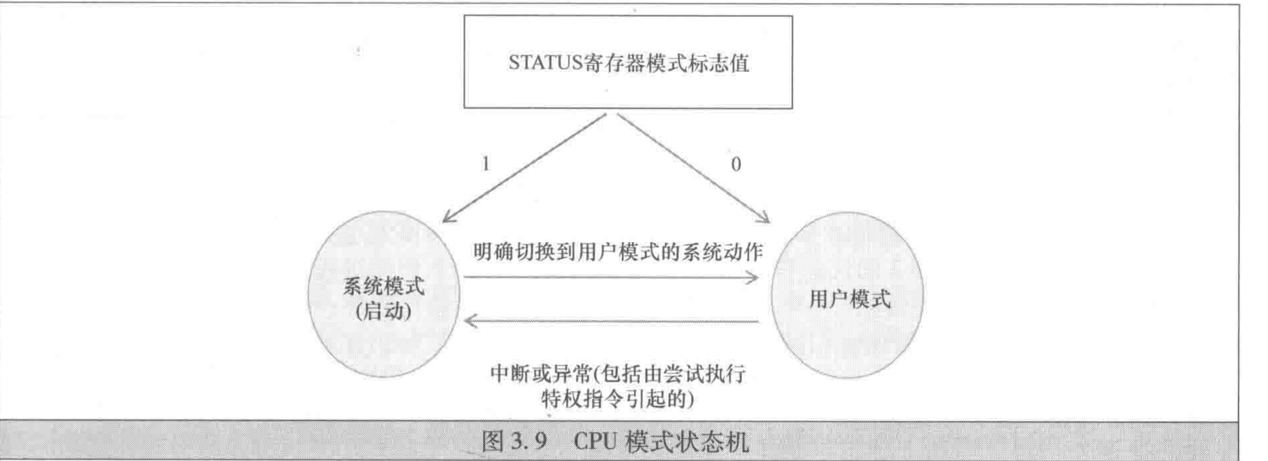


图3.9 CPU模式状态机

通过添加一个新的堆栈指针寄存器来进一步修改 CPU，所以现在有两个指针寄存器：一个（称为系统 SP）用于系统模式，另一个（称为用户 SP）用于用户模式。图 3.10 对这项修改进行了解释说明。指令中的寄存器名称（或相当于编码）有意对寄存器的选择保持不变，但是由于 STATUS 寄存器的模式标志，CPU 会知道要使用哪一个寄存器。

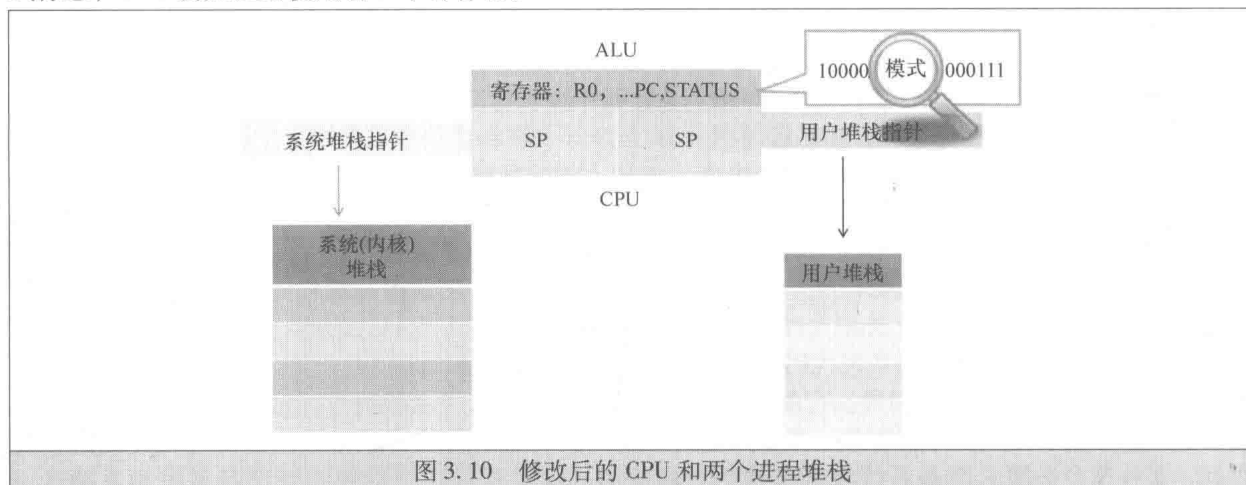


图 3.10　修改后的 CPU 和两个进程堆栈

借助这种变化，操作系统现在可以维护两个堆栈，一个用于执行系统代码，另一个用于执行用户代码。维护两个堆栈的概念起初可能看起来非常复杂，但它实际上简化了操作系统的设计。使用 Bach 著名专著[8]中所述的设计，在 UNIX 操作系统中已经取得了巨大的成功（细心的读者可能会有疑问，哪个堆栈用于保存中断处的 CPU 状态？答案是：系统堆栈。中断或异常发生时的第一件事是将 CPU 切换到系统模式，这将自动激活系统堆栈指针）。

接下来，介绍特权指令：只有当 CPU 处于系统模式时，这些指令才能成功执行。RTI、DISI 和 ENI 是这类指令集中的前三个。RTI 作为特权指令的原因是，它可以促使 STATUS 寄存器的恢复，因此可能导致转换到用户模式。正如之前假设的那样，这种转换可能只是由于系统代码的执行而发生的。

目前，DISI 和 ENI 是 STATUS 寄存器中标志操作的助记符。使用我们的 CPU，所有能够改变 STATUS 寄存器值的指令（例如，逻辑 AND、OR 或 LOAD）都被声明是具有特权的。类似地，所有能够改变系统堆栈指针的指令也被设计成具有特权，因为用户代码甚至无法引用系统 SP。

指令使用的背景环境是决定它是否需要特权的重要因素。

为此，假设所有处理 I/O 访问（存在于某些 CPU 中）的指令具有特权。将在 3.2.7 节中讨论特权指令的处理问题。首先，需要了解虚拟内存管理中的问题，这需要一整套新的特权指令。

3.2.6　虚拟内存——分段和分页 ★★★

现代虚拟内存的概念体现在两个方面：首先，使用虚拟内存，进程不必知道已经被分配的实际（物理内存）地址（相反，进程会“认为”每次运行时它和它的数据都占用着相同的内存地址）。其次，进程一定“认为”它可以访问所有的可寻址的内存，其数量远远超过了可用的物理内存。至少，期望具有 64 位地址的计算机拥有两个 EB（2×10^{18}字节）的实际内存是不常见的。

除了需要一些先进的硬件地址转换机制是实现这两个方面所通用的外，在其各自的需求上还存在着显著的差异。

先从第一个方面开始。在两个进程系统的情况下（如图 3.8 所示），分配给它们的分段显然也将具有不同的地址。然而，使用我们的 CPU，每个指令在内存中都有其预先定义的地址，并且内存也允许绝对寻址。因此，如果进程 2 的代码有一条指令 LOAD R1，30000，但该进程的数据分段从位置 5000 开始，那么必须以某种方式更改指令。一种方法是在加载时有效地重写程序，但这是不切实际的。更好的方法是采用一种行之有效的电子工程技巧，如图 3.11 所示。该图修改了图 3.2，在 CPU 和总线之间插入了一个称为内存管理单元（Memory Management Unit，MMU）的转换设备。该设备具有将逻辑（即虚拟）地址转换为实际物理地址的表（在现代 CPU 中，该设备实际上已经被集成到 CPU 中）。



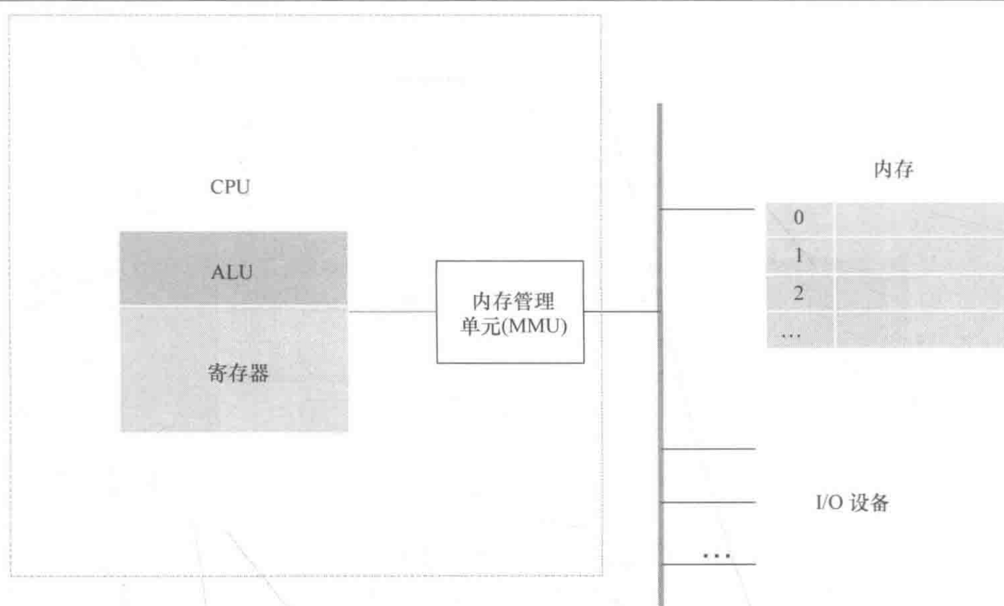


图 3.11 引入 MMU

假设已经被编程（或编译）的进程具有下述假设：

1）其代码从位置 x 处开始。

2）用户堆栈从位置 y 处开始。

3）系统堆栈从位置 z 处开始。

4）堆从位置 t 处开始。

其中每个都是一个连续的内存块，即一个分段。事实上，即使较小的分段在实践中也可以被考虑和使用。例如，把所有初始化代码放到内存中的一个地方是非常便利的（初始化完成后，还可以重复使用）。分段可以通过它们的编号进行枚举和引用。

在进程加载时，操作系统将其代码带入位置 x'时分别在地址 y'、z'和 t'开始处分配堆栈分段和堆内存分段。分段号是 MMU 表中的索引。该表中的一个条目是操作系统填充的分段基址寄存器（例如，x'或 y'）。MMU 具有将逻辑地址转换为物理地址的电路。因此，代码分段中的逻辑地址 a 被转换为 $a-x+x'$，它是分段基址寄存器 x'和代码分段内的偏移量（$a-x$）之和。这种转换可以通过硬件执行得非常快，所以它对时序的影响是可以忽略不计的。

MMU 还有助于确保进程不能读取或写入超出其分段范围之外的空间。因为，在 MMU 表中还有另一个条目：操作系统分配的分段内存的大小。继续上一个段落的例子，如果 X 是该代码分段的大小，则每个地址 a 均必须满足不等式 $a-x<X$。如果没有满足，则会生成异常。同样，MMU 电路执行与常数比较的操作也是非常快的，因此其影响也可以忽略不计。

图 3.12 总结了 MMU 地址转换过程。补充来说，MMU 在安全保护方面可以做更多的事情，并且通常也是这样做的。例如，可以在分段表中包含指示分段是否可执行的标志（从而防止意外“走过”数据，以及防止放置在数据分段中的恶意代码在后期被执行而发起攻击）。附加标志可以指示分段是否为只读、可以共享等状态，可以允许实现多样性的状态。例如，操作系统库可以被放置在面向所有进程为只读状态的代码分段中。

MMU 一般通过它的寄存器向外部提供访问。它几乎不需要解释为什么只有特权指令可以访问 MMU 条目。

内存虚拟化的第二个方面，就是创造“无限”内存的效果，相比来说这要复杂得多。我们将在这里对它进行简单的介绍，因为每种与操作系统相关的书籍[7,8]基本都对这个问题给出了详细的介绍。

首先，这里所采用的技术还解决了另一个“记忆碎片”的问题。由于多个进程在运行，因此对于操作系统来说找到一块连续的内存分段分配给其他的进程，基本是不可能的。不过，可能存在足够的“洞”（即没有被使用的相对较小的内存）累加在一起来提供足够的连续内存，这些“洞”会被合并到

连续的空间。

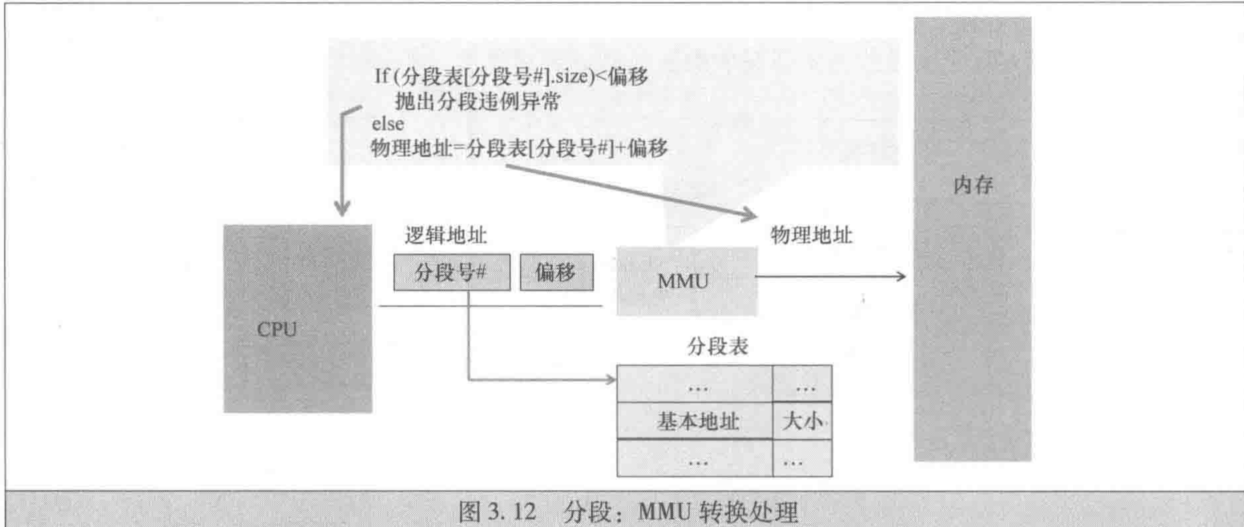


图 3.12　分段：MMU 转换处理

为了实现这一点，可以使用另一种类型的 MMU，它的工作原理如下。逻辑内存被看作是一个被称为“分页”的数组。所有的分页均具有相同的大小（总是将其选择为 2 的指数，所以如果最大可寻址内存为 n 并且分页大小为 f，则会有 n/f 个逻辑分页）。类似地，实际的物理内存被看作一个物理帧的数组，所有的物理帧均具有相同的大小，其大小与分页的大小相同。每个进程都有一个分页表，可以将每个逻辑分页映射到物理内存中的相应帧上。MMU 使用此表执行这种转换。

图 3.13 演示了一种情况，在 MMU 的帮助下，分散的内存块组成了在进程看来似乎是连续的分段（分页大小实际上是一个分割单位。换句话说，比分页大的内存碎片都被使用了）。

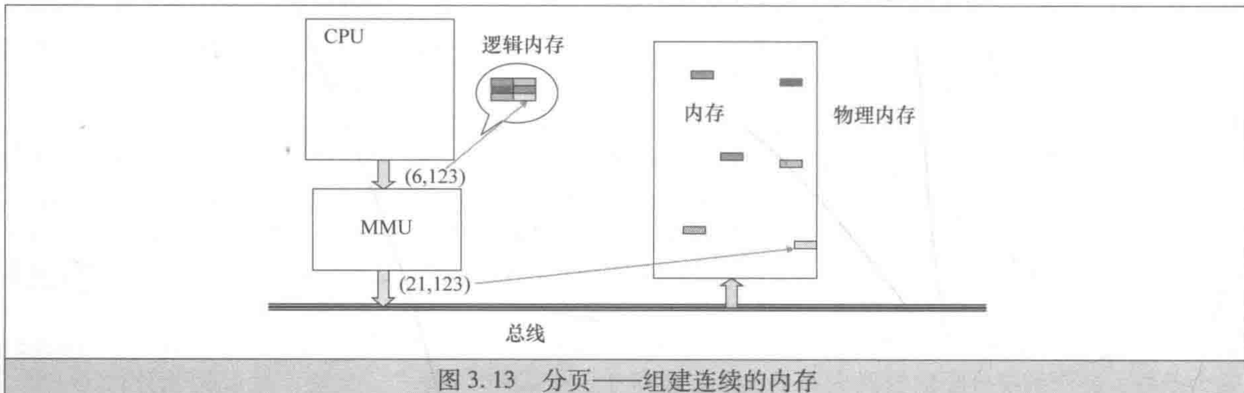


图 3.13　分页——组建连续的内存

假设一个进程需要访问它所认为的地址“5 × 分页大小 + 123”。如图 3.13 所示，该地址被解释为属于编号为 6 的分页，偏移量为 123。MMU 表将分页 6 映射到物理帧 21 上。最终，该地址被转换为实际的物理位置“20 × 分页大小 + 123”。

提供连续的内存是一个相当简单的功能。分页最巧妙的功能是它可以支持内存中的帧数比所有活动进程使用的总分页数少的情况。那样，即使单个进程的逻辑地址空间也可能比整个物理内存大。为了实现这一点，操作系统将进程内存映射到存储设备（通常为磁盘）。如图 3.14 所示，一些分页（称为驻留分页）保存在物理内存中，而其他分页则存储在磁盘上。当与非驻留分页对应的逻辑地址被进程引用时，必须将它写入物理内存。

当然，当内存中所有的帧都被占用时，它就变成了一个零和游戏：如果一个分页被带入内存，其他一些分页必将先被驱逐出去。通过由操作系统使用的分页替换算法选取需要逐出的分页。这种选择对于性能至关重要，因此在过去的几十年中人们对这一问题进行了大量的研究。如今，其结果可在任何操作系统的说明性文件中找到。

顺便说一句，实际的内存/存储设备并不存在什么孰优孰劣，也就是说，快速但相对较小且易失性的 RAM 和相对较慢但体积较大且持久的磁盘存储设备，都有自己的适用场景。在第 6 章中，将介绍并

讨论不同类型的存储方案，这些方案一起填补了两种极端之间的空白领域，并建立起了存储器的层次结构。

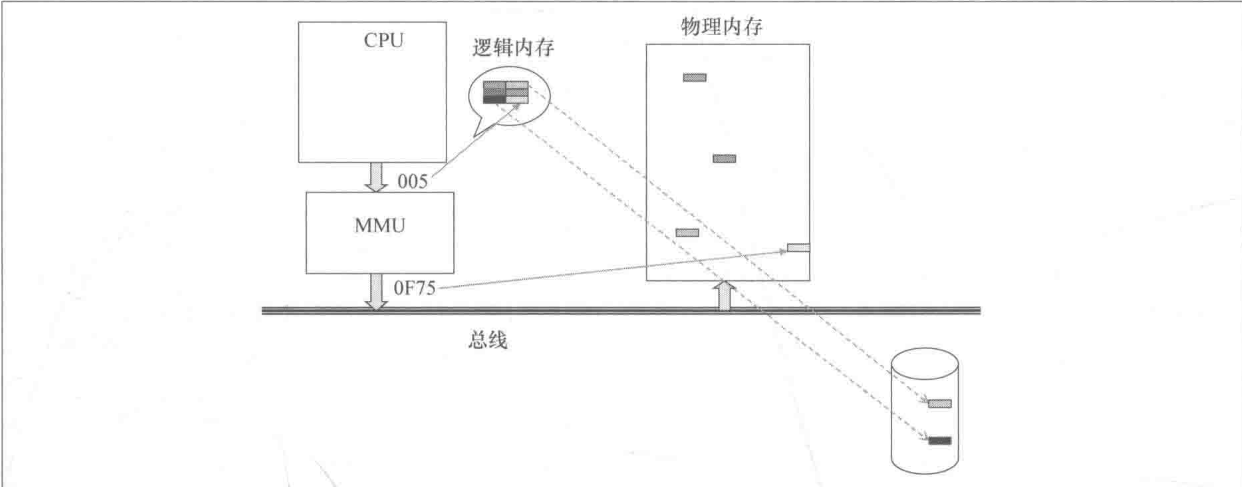


图 3.14 将分页存储在磁盘上实现“无限”内存的假象

图 3.15 描述的分页表提供了对转换实现的一些提示。虚拟地址最左边的比特位提供了逻辑分页号，其余的是偏移量，它保持不变。如果分页是驻留的，由表中相应的标志指示，则该转换就是直接完成的（并且相当快）。如果是非驻留的，MMU 就会生成异常，即分页错误。

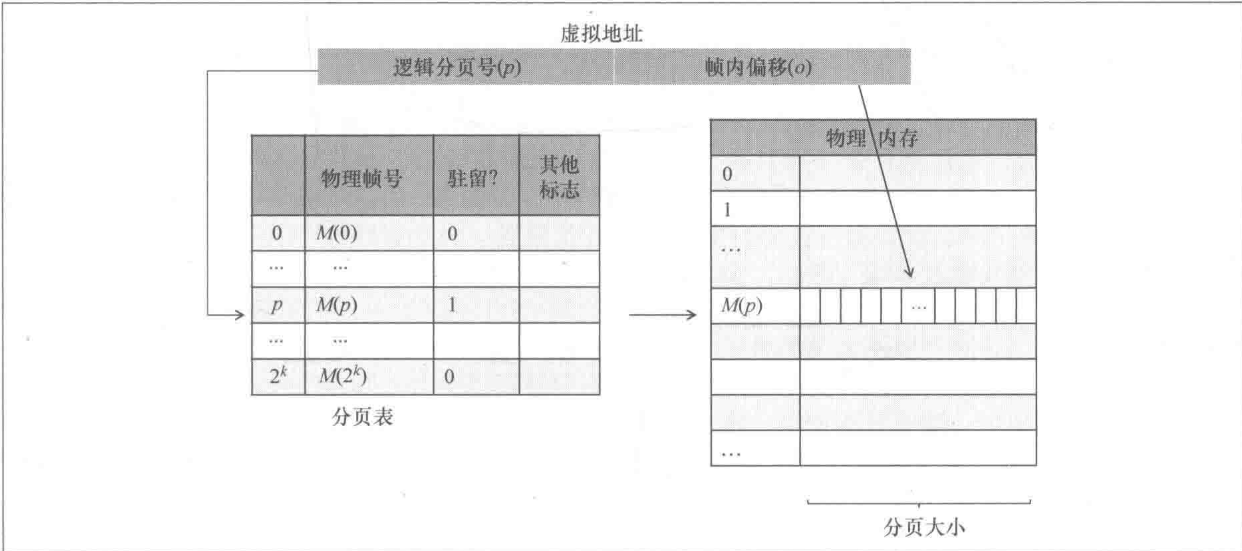


图 3.15 分页表和虚拟地址内存中转换

要处理分页错误，操作系统需要找到要逐出的分页（分页替换算法使用分页表中存储的其他的标志）。然后，要驱逐的分页可能会被写入磁盘。最后，等待被引用的分页将会从它在磁盘中的位置读入到内存帧中。这一过程需要完成很多的工作，但是借助于分页的特点，它可以相当高效地完成。虽然虚拟内存比真实内存慢得多，但复杂的基于启发式的算法使其不会慢得太多。同时，值得注意的是，进程分页表可以增大，远大于 MMU 可以填充的程度，从而在这种类型的转换中引入额外的机制。

此外，与分段表一样，分页表也采用了内存保护，即一个分页可能只是可执行的、只读的，或它们的任意组合。而且，与分段表的情况一样，能够被调用更改分页表的指令都是具有特权的。

在本节中，最后需要注意的是：分段可以与分页相结合。例如，每个分段都可以有自己的分页表。

3.2.7 特权指令的处理选项和 CPU 循环的最终近似 ★★★

一个重要的问题是，CPU 在用户模式下遇到特权指令会做什么（特权指令如何出现在用户程序?

当然，合法的编译器不会产生这样的程序，但是出现这样的情况，有可能是汇编程序人员的失误或出于恶意的企图。此外，在完全虚拟化中，也有可能出现这样的情况——对于这一问题将在3.3节中讨论）。

所有现有的CPU在处理这种情况时，都会采用以下两种方法之一。当CPU在用户模式下遇到特权指令时，它要么生成异常，要么完全忽略这条指令（即跳过它，不执行任何操作，只是浪费了一个周期）。

无论哪种方法，它都确保了用户程序不会通过执行特权指令造成任何破坏。但是，第二种情况会导致虚拟化方面的问题。稍后在研究虚拟机管理程序时，再讨论这个问题。

通过设计，我们开发的小CPU㊀在用户模式下遇到特权指令时会生成一个异常。通过这种方式，CPU提供了一个明确的机制，指示问题的发生。它还可以帮助改善安全性，就安全考虑而言，检测对系统控制权的获取企图，对防止其他来自同一来源可能更为复杂的尝试是至关重要的。忽略“奇怪”的指令对安全防护没有帮助。将控制权从包含这类指令的程序中移除。

因此，我们得到了第三个，也是最后最近似于现代的CPU循环，如图3.16所示。与早期版本相比，第一个主要的变化是，每个指令都是在一开始就进行检查的。如果不能识别，或者具有错误的参数，或者不适用于当前的模式（即如果它是特权指令，而CPU正在用户模式下运行），那么就会出现异常。另一个变化是，当CPU开始处理中断或异常时，它会切换到系统模式，并切换到系统堆栈（至于时序方面，STATUS寄存器当然是在CPU模式切换之前被保存的，所以CPU可以返回到中断时的模式）。同样，寄存器被保存在系统堆栈，而不是用户堆栈，这是另一种安全措施。通过良好的设计，用户代码既不能接触，甚至也无法看到这些系统数据结构。

```
While TRUE
  {
     通过PC取得指令指针:
     If 指令有效 AND 指令适用于当前模式 AND 参数对操作有效
         {
            推进PC指向下一条指令;
            执行指令;
         }
      else
         生成适当的异常;
     If （异常#x已经出现） OR （中断#x已经出现） AND 中断激活
       {
          将STATUS寄存器和 PC保存在系统堆栈(@SP );
          切换到系统模式;
          PC = 中断向量[x];
       }
  }
```

图3.16 最终版的CPU循环

3.2.8 更多的操作系统内容★★★

到目前为止，所创建的CPU可以支持虚拟化，因为其上运行的进程可以“认为”自己拥有整个CPU（即使是较慢的CPU）、所有的I/O设备，以及无限的统一寻址空间。物理机器由操作系统软件控制，该软件创建并维护用户进程。操作系统提供的服务是操作系统的特点。

我们已经解决了CPU访问和内存管理等服务问题。其中，后者延伸到了长期存储器——磁盘文件系统的建立与维护。磁盘只是由操作系统管理的I/O设备之一。显示器、键盘和打印机是其他设备的例子。另一个例子是网络驱动器。进程只能通过操作系统访问设备。

㊀ 这样的设计决定是在开发M68000系列处理器时做出的[9]，事实上，我们的示例CPU体现出了该机器的几个功能。

虽然进程似乎拥有 CPU、内存和设备，但它还需要知道其他进程的存在。为此，所有现代操作系统都为进程间的通信提供了一种机制，它可以让进程交换信息。结合数据通信，该功能可以实现在不同机器上运行的进程之间进行通信，从而为分布式计算奠定了基础。

这些服务是以层为单位构建的，每层服务于其上方的层。如果在数据网络服务之上实现文件系统服务，则可以将进程的文件系统映射到与其连接的不同计算机的磁盘上，进程无须知道设备的实际位置和具体文件记录的位置。

进程通过系统调用请求操作系统服务。对程序员来说，这种调用只是一个过程调用。不可避免地，相应的过程可能会包含陷阱指令，导致进程执行中断。图 3.17a 所示包含了一个示例 Service_A 例程的伪代码。它所做的就是在用户堆栈上推送 Service_A 助记符，并通过 TRAP 1 将控制权传给系统。



图 3.17　系统调用处理

如图 3.17b 所示，当 TRAP 1 异常被处理时，该进程将恢复。处理程序读取进程堆栈的顶部以确定服务 A 已被请求，然后调用 Service_A_System 例程。如图 3.17c 所示，后者以系统模式运行，可能会调用其他的系统过程。运行完成后，返回到图 3.17b 所示的异常处理程序，该处理程序在最后通过 RTI 指令将控制权返回给用户程序代码。读者可能还记得 STATUS 寄存器（带有指示用户模式的模式标志）的值在中断时被保存。而这种向用户模式的切换只能通过恢复该值来实现。

重要的是要了解，用户进程会经历一连串的在用户程序代码与系统代码之间交替的指令。然而，用户程序不知道，也不能控制系统数据结构或其他进程的数据（当然，除非其他进程有意共享此类数据）。

需要简单指出的是，当服务例程启动 I/O 操作时，通常不会等待这项操作完成。相反，它会把进程停放在该设备特定事件的队列，并将控制权传递给调度程序，以便其他进程能够运行。为此，如果操作系统确定该进程的量子或时间片已经耗尽，那么它将发起一个时间片中断，该中断可能会产生完全相同的中断进程处理过程。通过这种方式，操作系统可以确保 CPU 能够被公平地分配给所有的进程。

在这一点上，可以适当地解释术语“进程”和“线程”之间的区别。进程会被分配资源：它的内存空间、设备等。最初，这些资源只与单个控制线程相关联。后来，进程这个词变得只和地址空间相关，而不是执行的线程。因此，多个任务（线程）可以在进程中执行，共享相同的资源。有时，线程也称为轻量级进程，因为线程的上下文切换（不需要重新加载分页表等）与进程相比需要调用的过程要少得多。不幸的是，这个术语没有被一贯使用。

操作系统的演变受到两个因素的影响：应用需求和硬件功能（因此也是硬件价格）。当需要考虑实现细节时，仍然存在一个灰色区域，使人们无法假定给定的功能是放在硬件上实现还是放在操作系统中实现。

数据网络的发展使操作系统的功能可以分布在网络中的不同节点上。这实际上是 20 世纪 90 年代到 21 世纪初的一个发展趋势，直到后来云计算的出现，延续了这一趋势。与此同时，这种趋势似乎已被扭转［工作集中在一台（虚拟的）机器上］，而在理念上仍在前进（虚拟机可以轻松地通过网络从一台机器迁移到另一台机器上）。

在整个操作系统的发展史上，一般的安全性问题（以及较小的隐私问题）已经得到不断的解决。有意思的是，即使在它根本不是个问题的时候；也就是在没有网络的情况下，甚至在很少人才能接触到计算机的时候也是如此。操作系统的历史通过人类整体历史的发展表现已经证明了：即使有最好的

想法和最好的设计，也不一定会成功。为此，麻省理工学院的多路信息与计算服务（Multiplexed Information and Computing Service，Multics）[10]操作系统的功能集从未完全出现在无处不在的商业产品中，尽管 Multics 已经影响了几个操作系统的设计，尤其是 UNIX 操作系统。

Multics 十分看重安全方面，对待安全问题非常认真。罗伯特·格雷厄姆（Robert Graham），Multics 的贡献者，一位具有远见卓识的人，曾在他 1968 年的开创性的论文[11]（其中非常准确地预测了几乎半个世纪之后计算机信息领域的发展）中写道：“用户社区必定会有不同的兴趣。事实上，这些用户社区中可能会包含具有商业竞争力的用户。这套系统将可以用在很多的应用方面，其中敏感数据（如公司工资单记录）将需要存储在系统中。另外，社区用户希望彼此分享数据和程序。甚至一组用户将在同一个项目上协同工作……最后，会有信息处理实用程序管理提供的公共程序库。事实上，这样一个系统的主要目标是为许多不同的用户提供灵活但可控的共享数据和程序的访问”。

格雷厄姆的设计，已经在 Multics 中得到了实现，他在这一设计中设计了保护环。我们从一个简单的方案开始，如图 3.18 所示。回顾一下 3.2.6 节中讨论的分段表，可以进一步细化分段的访问权限。

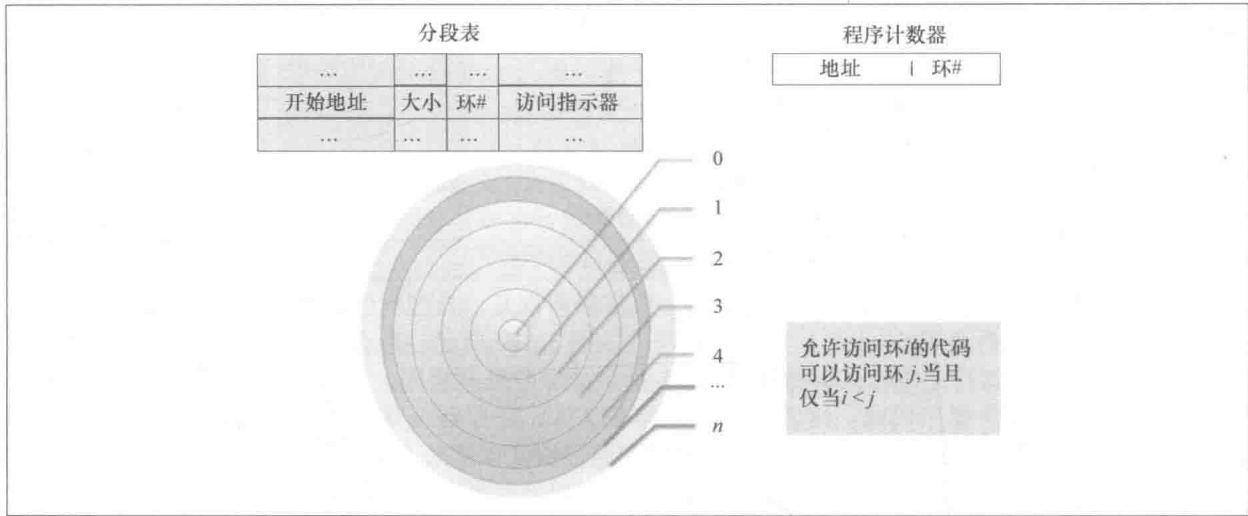


图 3.18　格雷厄姆的安全环（硬件支持）

专注于每个分段而不是每个分页的保护方案的确定是经过深思熟虑的。在格雷厄姆的文章中清晰写道：“分段是用户识别信息的逻辑单元”，而“分页是对系统存储管理有用的信息单元，因此对用户是不可见的”。这种保护方案的思想是根据军事原则“根据许可级别确定知密范围”而产生的。拥有给定许可级别的人可以访问低于该许可级别的信息，相反这不行。因此，将权限级别表示为一组同心环。最小的环（环 0）具有最高的权限级别（这一点的图形意义在于，权限级别越高，被授予访问这一级别的权限的组越小，所以用较小的圈子表示具有较高权限级别的环）。

格雷厄姆在本章参考文献［11］中提出了两个模型。第一个比较简单直接；第二个更为复杂，因为它提供了优化。

两个模型都需要更改分段表。两个模型的访问指示字段包含了 4 个独立的标志位：

1）用户/系统标志——指示分段是否可以在用户模式下访问。

2）读/写标志——指示分段是否可以写入。

3）执行标志——指示分段是否可以执行⊖。

4）故障/无故障标志——指示访问分段是否会导致异常，无论在何种情况（额外的保护级别，覆盖所有其他的标志）。

PC 寄存器增加了指示代码权限级别的环号。另外，每个分段表条目都增加了一个对环结构的引用，但是这里的模型不同。

在第一个模型中，每个分段现在都被分配了环号，指示它的权限，如分段表条目所示。需要注意

⊖ 在本章参考文献［11］中有一些细节反映了这些标志值具体组合的含义。例如，相应的标志值为“读取和执行”则用来表示代码不能自行修改。

的是，此方案会导致不相交的分段集被分配不同的权限。

这里考虑一个代码分段。如果该分段访问环值是 i，分段内的代码只能访问分段环值为 $j>i$ 的分段（当然，这只是第一道防线。通过访问指示标志的值可以对访问做进一步的限制）。跳出分段需要系统的特殊干预，因此它们都会引起异常。

虽然这个模型是有效的，但格雷厄姆表示，如果放松环不相交的要求，允许以调用者的权限级别（但不能更高）执行共享例程，则可以使它更有效。通过这个结果可以得到第二个模型，如图 3.19 所示。

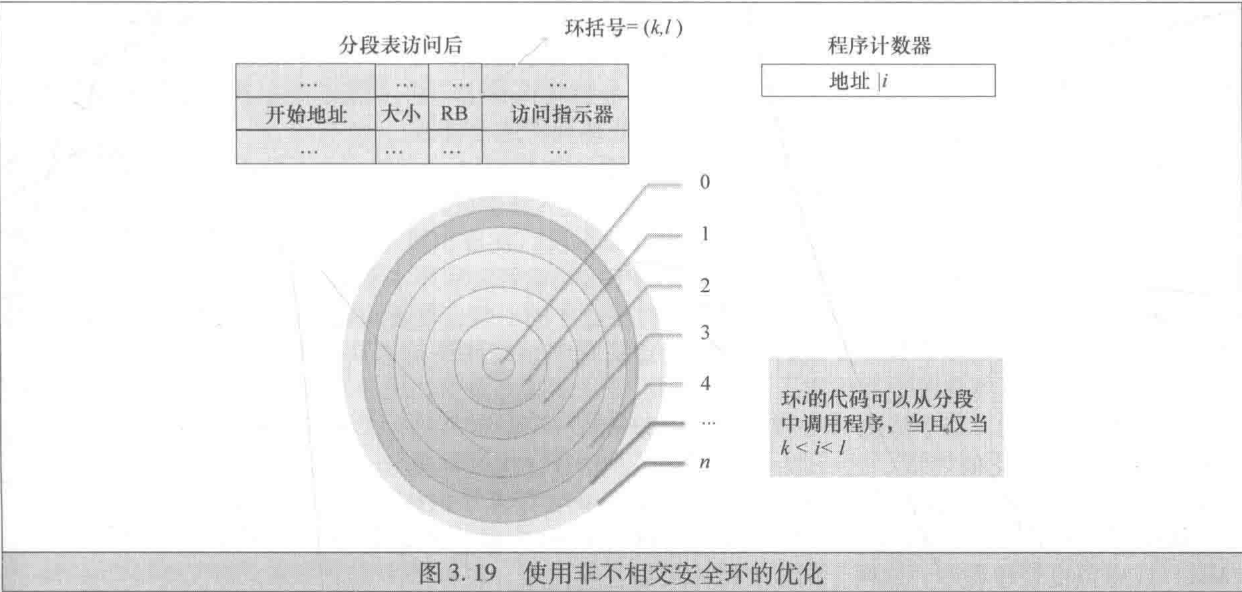


图 3.19　使用非不相交安全环的优化

这里，分段表条目中的环号被替换为两个数字，分别表示分段环括号的下限和上限。分段中有意成为函数库的所有代码被分配到括号中的连续分段上。因此，当以权限级别 i 执行的过程调用一个访问括号是（k，l）的过程，只要 $k<i<l$，就不会发生异常。超出括号限制的调用会导致异常。

到目前为止，只考虑代码分段之间的控制转移，但是环括号的使用可以进一步延伸到以下的数据分段。如果一个数据分段的环括号为（k，l），那么以权限级别 i 执行的指令可以：

1）写入该分段，只要它是可写入的并且 $i\leqslant k$（即指令的权限级别比该分段高）。

2）读取该分段，前提是 $k<i\leqslant l$。

如果 $i>l$，那么指令完全不能访问该分段。

Multics 设计了 8 种保护环，环的数目受到硬件的限制。早在 20 世纪 80 年代，通用数码公司（Data General）的 Eclipse MV/8000 CPU 已经在 PC 内设计了一个三位的环指示器（相关内容请参见本章参考文献［12］）。

这些发展都预见到了将来，曾经的这些发展正是今天正在发生的。从虚拟化的角度来看，对安全保护的重燃兴趣使得许多计算机科学家开始怀念起了 20 世纪 70 和 80 年代。

在这一点上，总结前面的内容，可以归纳如下：

1）进程作为一个单元，有自己的虚拟资源世界：CPU、“无限”的内存和所有的 I/O 设备。

2）进程依赖于操作系统——操作系统对物理机器进行管理，确保所有进程得到公平的对待。

3）操作系统内核是唯一可以执行特权指令的实体。内核作为处理中断或异常（包括陷阱）的结果加入。

4）物理机器可以执行多个进程。

5）进程可以感知到其他进程，并且可以与它们进行通信。

下一节将介绍如何将其引入下一级抽象，以便开发图 3.1 所示类型的虚拟机。

3.3 虚拟化和虚拟机管理程序

先来回顾一下我们的目标。

通过引入虚拟化，首先是为了节省成本，主要是在空间、能源和人员方面的成本，我们可以在一台机器上运行多台机器。其中，一个越来越重要的方面是环保问题：一台机器通常要比多台机器使用的电量更少。

虚拟化的另一个目标是提高 CPU 的利用率。即使使用多重处理，CPU 几乎也从未被充分利用。在“云计算”这个词出现之前，我们可以看到，运行 Web 服务器的机器 CPU 的利用率比较低。现在，服务器可以被组合放到曾经只能运行单个服务器的一台物理机器上，多个服务器在其各自的虚拟机上运行，可以使它们在节省能源和硬件开支的同时，更好地利用这台机器。

其他目标包括低成本地克隆计算环境（如服务器）进行调试。例如，为了应对增加的负载，迁移机器；为了特定的目的隔离设备（如服务器），而无须投入新的硬件。

就安全性而言，隔离是分析未知应用程序的有效手段。在虚拟机上测试程序，可以将其造成的安全风险隔离到这台虚拟机上。这个绝妙的功能取决于虚拟机管理程序本身，这将在本章最后一节介绍。

在我们看来，云计算作为提供计算服务实用程序业务的出现也是虚拟化的结果（而不是目标）。当然，它并不是由虚拟化技术单独提供的，而是由虚拟化和快速网络技术共同提供实现的，但是我们认为虚拟化技术在这里发挥了重要作用。

先从一个一般的、基于传统 CPU 架构的假设开始，即操作系统内核本身不能创建独立的虚拟机。随着最近 CPU 虚拟化的发展，这一假设不再成立。并且，将回顾基于内核的虚拟机（Kernel - based Virtual Machine，KVM）管理程序，其中正如其名，它确实是基于内核的一个特例。

从虚拟化开始，负责创建和维护虚拟机的软件被称为虚拟机监控器（Virtual Machine Monitor，VMM）或虚拟机管理程序。这两个词在文献中可以互换使用。在本书的其余部分，将仅使用后一个词，即虚拟机管理程序。虚拟机环境由虚拟机管理程序创建。

3.3.1 模型、需求和问题 ★★★

如本章参考文献［13］所述，用于解决在虚拟机上进程执行所需资源的经典抽象模型，已经出现了40多年。在这个模型中，采用了两个功能，一个对于操作系统是可见的，而另一个功能仅对虚拟机管理程序可见。前一个功能将进程 ID 映射成（虚拟）资源名称；后一个功能则将虚拟资源名称映射为物理资源模型。这种映射是递归的，允许在物理硬件之上的多个虚拟机层上进行。每当发生故障（即引用了未映射的资源），异常会导致将控制传递给下一层的虚拟机管理程序。该模型为已经为虚拟化的系统研究和实现奠定了基础。

一项重大的后续研究工作[14]使其作者的名字“杰拉尔德·波佩克（Gerald Popek）博士和罗伯特·戈德堡（Robert Goldberg）博士”永远同虚拟化相关联，即“波佩克与戈德堡虚拟化需求”，并且这份需求已经成为了经典，值得人们深入研究。

首先，这篇论文的主要任务是陈述和演示“计算机托管虚拟机管理程序必须满足的条件”。为此，作者设计了一种公式化的机器模型，并在数学上证明了他们的结论。该模型是第三代 CPU 架构的抽象，IBM 360、Honeywell 6000 和 DEC PDP - 10 都是这一代 CPU 的具体实例，在本章前面已经设计的 CPU 也是这样的。该模型不考虑中断，因为这些对于其论述的主题不是必需的，但是对陷阱进行了完全的建模。

这篇论文概述了虚拟机管理程序（作者使用 VMM 一词）的 3 个基本特征：“首先，VMM 为与原始机器相同的程序提供了一个环境；其次，在这种环境下运行的程序在最坏的情况下只显示出微小的速度下降；再次，VMM 完全控制系统资源”。

第一个特征，通常被称为等价需求，即虚拟机上运行的程序必须产生与它在真实机器上运行相同的结果。

第二个特征是效率需求：将“硬” CPU 能够直接执行的指令从软件模拟的 CPU 指令范围中排除。作者认为与软件模拟相比在真实 CPU 上运行其性能具有“统计优势”的这部分指令，必须由真实的处



理器直接执行。

第三个特征被转化为以下的资源控制需求：

1）虚拟机上运行的程序不能访问虚拟机管理程序未明确分配给它的资源。

2）虚拟机管理程序可以从虚拟机中取走以前分配给它的任何资源。

现在，可以介绍本章参考文献［14］中所述的正式模型了（这部分内容可以跳过，不会影响对其研究结果和原理的理解）。该模型基于对一个四元组 $S=<E,M,P,R>$ 机器状态的定义，其中 E 为内存状态，M 为 CPU 模式（即用户模式或系统模式），P 是 PC 寄存器的值，R 是“重定位寄存器”（在 CPU 的情况下，这实际上是整个 MMU，但是早期的第三代 CPU 实际上只有一个分段，因此只有一个重定位寄存器）。该模型使内存位置 $E[0]$ 和 $E[1]$ 称为保存 $<M, P, R>$ 三元组的有效堆栈。指令 i 可以被看作状态空间到自身的映射，因此可以写成

$$i(E_1,M_1,P_1,R_1)=(E_2,M_2,P_2,R_2)$$

如果存储保持不变，则该指令被称为陷阱，除了堆栈的顶部：

$$E_2[0]=(E_1,P_1,R_1);(\text{保存的状态})$$
$$(M_2,P_2,R_2)=E_2[1]$$

如果内存访问越界时陷入陷阱，则该指令被称为内存陷阱。

因此，可将指令进行如下分类：

1）如果指令没有内存陷阱，则将其称为特权指令，但只有在用户模式下执行时，才会陷入陷阱。

2）如果指令尝试更改可用内存量或影响 CPU 模式，则将其称为控制敏感指令。

3）如果指令的执行取决于控制敏感指令设置的控制值，则将其称为行为敏感指令。作者提供的例子是 IBM 360 加载真实地址（Load Real Address，LRA）指令，它是位置敏感的；以及 PDP－11/45 从前指令空间移动（Move From Previous Instruction Space，MFPI）指令，它是模式敏感的（后一条指令是该机器的四条指令之一，在当前堆栈上复制前一地址空间中操作数地址上的数据，如 CPU 模式所示。我们定义的 CPU 没有这样的功能）。

4）如果指令是行为敏感或控制敏感的，则将其称为敏感指令。否则，将其称为无害指令。

果然，敏感指令定义的规范化涉及相当多的符号复杂性，作者称之为“不幸的符号表达式的切线”。基于这种符号表示开发的理论使我们能够正式证明这一结论，即如果所有敏感指令都是特权指令，那么就可以构建一个虚拟机管理程序，这是这篇论文的主要结论。

为此，该虚拟机管理程序的构建如图 3.20 所示。

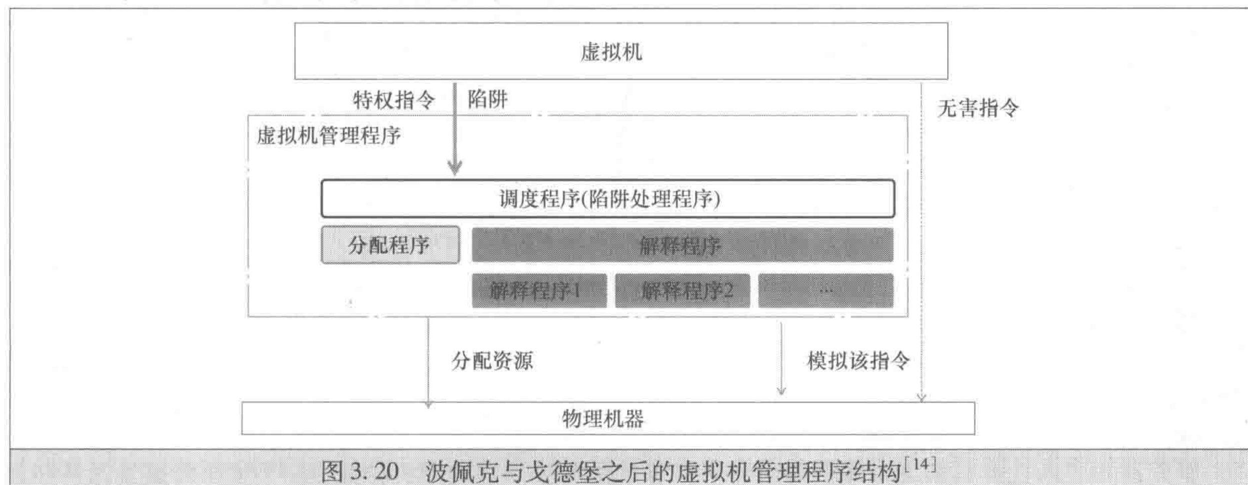


图 3.20　波佩克与戈德堡之后的虚拟机管理程序结构[14]

虚拟机执行的代码中的所有无害指令都由物理机器的 CPU 立即执行。尝试执行特权指令会产生（定义好的）一个陷阱，调度程序是它的服务例程。如果一个指令需要资源分配，调度程序将调用分配程序。最终，被陷指令由解释程序模拟，解释程序只是一个模拟例程库，每条指令一个。

本章参考文献［14］中的模型没有专门关注分时问题，尽管作者指出，CPU 本身也是分配程序所考虑的可分配资源。

的确，今天的虚拟机管理程序就是在分时环境下工作的，如图 3.21 所示。

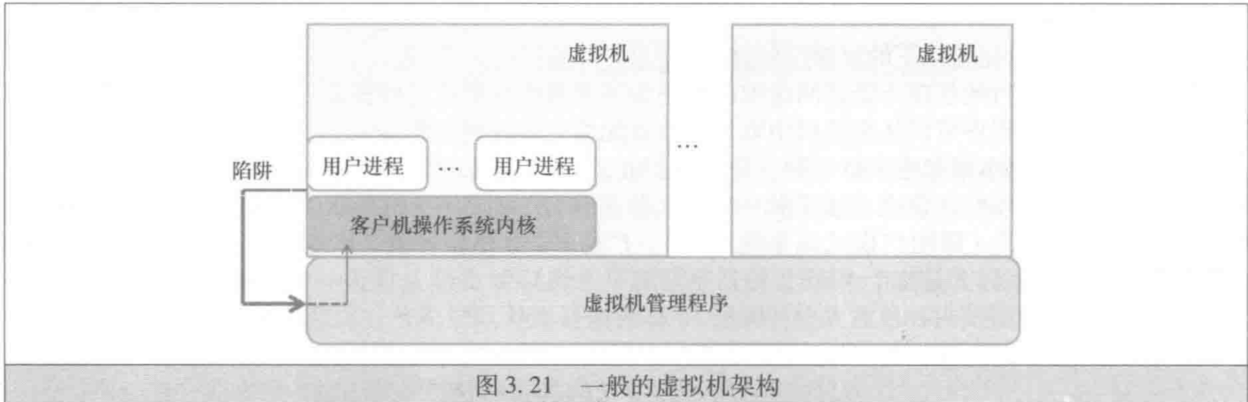


图 3.21　一般的虚拟机架构

在这种架构中，虚拟机管理程序负责维护虚拟机并根据三大虚拟化需求为每个这样的虚拟机分配资源。

每个虚拟机都运行自己的操作系统（称为客户机操作系统），处理用户进程。虚拟机管理程序获取所有中断和异常。在后者中有与系统调用相对应的陷阱。

一些中断（如处理多个机器分时的闹铃中断）是由虚拟机管理程序本身处理的，其结果是，已经超过其时间量子的虚拟机会被停止，并将 CPU 分配给其他的虚拟机。

专用于虚拟机的所有中断和异常（包括从本机用户进程发出的系统调用）都将传递给客户机操作系统。

但虚拟机管理程序运行在哪里？到目前为止，假设它直接在物理机上运行。这样的虚拟机管理程序称为 1 型（或有时称为裸机）管理程序。相比之下，在操作系统之上运行的虚拟机管理程序称为 2 型管理程序。这种分类如图 3.22 所示。



图 3.22　1 型和 2 型虚拟机管理程序

在本章后面的具体案例研究中，再来回顾这种分类法。

到目前为止，我们一直专注于 CPU 虚拟化的问题。不用说，其他虚拟化要素（例如，I/O 设备访问的虚拟化）并不是无足轻重的，特别是在实时操作时。就内存虚拟化而言，尽管操作系统已经在进程级别解决了这一问题，但由于虚拟机管理程序需要维护其自己的分页表“影子”副本，所以这仍然是一个非常重要的问题。

尽管如此，最难的问题是波佩克与戈德堡术语中所指的不可虚拟化的 CPU 带来的。其中，使用最广泛的一个就是 x86 处理器。

3.3.2　x86 处理器和虚拟化 ★★★

服务器和个人计算机的主力处理器同样都起源于原来的 Intel 8086 CPU，其 1979 年的型号（8088）由 IBM 个人计算机率先出品。当时，8088 突出重点的部署策略（即聚焦个人计算）可能并没有关注虚拟化（有什么可以在为单个用户设计的机器上进行虚拟化的呢?）。

相比之下，摩托罗拉 M68000 处理器（8088 的对手，还有苹果 Macintosh、Atari 和 Amiga 计算机的 CPU，以及 AT&T UNIX PC 工作站）的架构适合于从零开始的虚拟化。为此，M68000 处理器只有一个不可虚拟化的指令。

该指令允许在用户模式下执行，将 STATUS 寄存器的值传送到通用寄存器。在本章前面设计的 CPU



中，其等效于 LOAD R1，STATUS。该指令在波佩克与戈德堡分类法中是行为敏感的，因为它允许用户程序发现其正在运行所在的模式。这里存在的风险如下：假设虚拟机正在运行在安全性考虑方面精心设计的操作系统。该操作系统的一个特点是，一旦发现其内核代码在用户模式下运行，就立即使其停止执行。由于 STATUS 寄存器具有模式标志，所以所有实现此功能的操作系统将停止。

为了让 M68000 能够装配到可以运行多个操作系统的微型甚至小型计算机的目标，摩托罗拉快速地解决了这个问题。1982 年发布的 M68010 处理器，使这条指令当在用户模式下执行时会陷入陷阱，就像所有其他指令一样㊀。

然而，最终接管计算领域的是 x86 CPU，而不是 M68010。但是，尽管 x86 CPU 的指令集通过精心的设计拥有了后向兼容的思想，但在它的指令集中仍然包含不可虚拟化的指令。例如，32 位处理器 Intel Pentium CPU 的指令集中，有 17 条指令㊁被确定为不可虚拟化的指令[15]。这些指令分为 3 类（实际上，对问题的理解至关重要的还是这些划分的类别，因为在处理器之间指令不同的时候，这些分类却是大体不变的）。我们来看看 Intel Pentium 处理器中的指令类别。

第一类包含读取和写入全局和局部分段表寄存器以及中断表寄存器值的指令，其中所有的这三个分别指向它们各自的表。一个主要的问题是该处理器只有一个寄存器，这意味着需要为每个虚拟机来复制它们。因此，需要映射和特殊的转换机制来为其提供支持。由于这些机制必须在每次读写访问时运行，所以对性能的影响是显而易见的。

第二类包含将 STATUS 寄存器的部分值复制到通用寄存器或内存（包括堆栈）中的指令。这里的问题与前面描述的 M68000 中的情况一样：虚拟机可以发现其运行时所处的模式。

第三类包含依赖于内存保护和地址重定位机制的指令。要介绍这一类，需要引入 x86 指令集中的代码权限级（Code Privilege Level，CPL）。有 4 个权限级，按照权限下降的顺序为从 0 到 3，如图 3.23 所示。因此，以较高权限级运行的代码可以访问以较低权限级运行的代码可用的资源。所以，以 CPL 0 运行可以访问 CPL 1、CPL 2 和 CPL 3 对应的所有资源（回想起来，为了反映更高权限的定义，可以将其更精确地描述为，CPL 0 在外环，CPL 1 在下一环，等等，但是长远的做法是按照图中描述的方式来对这些环进行分配）。两级操作系统的做法是，以 CPL 0 执行系统代码，以 CPL 3 执行用户代码。

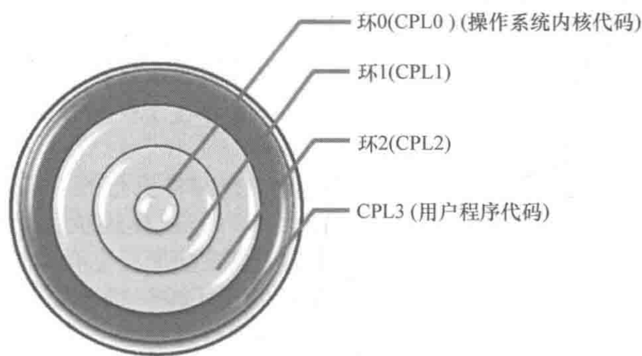


图 3.23 Intel 权限级环

回到第三类不可虚拟化指令的讨论上来，其中包含了引用存储器保护系统、内存或地址重定位系统的指令。当然，这一类相当多，因为即使是基本指令（类似于 LOAD 指令）也有需要进行分段寄存器访问的变量。

但是从分段表检查访问权限或分段大小的指令，需要确保当前的权限级别和请求的权限级别都要比分段的权限级别大。这样就会造成执行问题，因为虚拟机不会以 CPL 0 去执行。

包括过程调用在内的跳转会产生更多的问题，部分原因是它们比本章前面提到针对简单的 CPU 所

㊀ M68010 处理器具有多个辅助虚拟化的属性，其中包括配置中断向量位置（之前始终保持在低地址存储器中）的能力和处理实现虚拟内存所依赖的“总线错误”异常的能力，使得导致它的指令在内存被“修复”之后能够继续运行（而不是从头开始运行）。

㊁ 通常来说，这样的指令的数量是 17，但是有一些观点反对这个具体的数字，这些观点认为这个数字是根据副作用（不能支持虚拟化的效果）计数得出的。最有效的方法应当是查看这些受影响的指令的类别。

定义的 JUMP 和 JPR 指令也复杂得多。在 x86 架构中，有两种类型的跳转：近跳和远跳。每一类又可进一步细分为两个类别：跳转到相同的权限级别和跳转到不同的权限级别。同过程调用类似，其中还有任务切换。如本章参考文献［15］所述，不同权限级别的调用和任务切换的问题是通过比较不能在用户级别正常工作的操作权限级别引起的。

还有另一类不可虚拟化的指令，即那些只能在系统模式下执行的指令，当尝试以用户模式执行它们时，它们既不会执行，也不会陷入陷阱。这些指令有时也被称为不良指令。它们所带来的问题过于明显，这里不再讨论。

我们将“完全虚拟化”定义为一种与单独物理 CPU 所提供的没有太大区别的计算环境。完全虚拟化的一个重要特征是，不需要修改在该环境中创建的虚拟机所运行的操作系统。显然，完全虚拟化不能在未修改的 x86 CPU 上实现。

除了在每一条指令中以软件模拟 CPU（这是不切实际的）之外，处理不可虚拟化的指令还有两种主要方法，将在下一节中讨论。最终，就软件而言，这一切都是可以做到的。

然而，随着 x86 不断扩展它的虚拟化功能，已经有可能进入纯粹硬件辅助的虚拟化途径。这里唯一的问题是 x86 处理器由多家公司［Intel、Cyrix、Advanced Micro Devices（超微半导体公司）和 VIA Technologies（威盛电子）等］制造，尽管基本指令集的行为在制造商之间是标准化的，但虚拟化机制却不是。Intel（VT－x）[16]和 AMD（Pacifica 以及后来的 AMD－V）[17]向虚拟化扩展的早期方法，在本章参考文献［18］中进行了很好的介绍和比较。

虽然制造商之间的扩展不同，但他们共同的功能是提供新的客户机操作模式，其中软件可以请求某些指令被捕获（以及访问某些寄存器）。其中伴随着以下两个功能（如本章参考文献［19］中总结的），通过它们实现虚拟化。

1）硬件状态转换。在进入和退出客户机操作模式的过程中触发硬件状态转换，它可以改变影响处理器操作模式的控制寄存器和内存管理寄存器。

2）退出原因。退出原因反映了客户机操作模式转换的相关信息。

这些功能使完全虚拟化得以实现。在回到这一点之前，先来看一下解决不可虚拟化 CPU 问题的通用方法。

3.3.3 不可虚拟化 CPU 的处理 ★★★

所有解决方法的共同点在于修改原始的二进制代码。差异在于代码修改的时间：是在运行时修改，还是通过全局预处理在运行之前修改。

采用二进制重写，虚拟化环境软件可以近乎实时地检查即将执行的一组指令。然后，用陷阱替换原始指令，将控制传递给虚拟机管理程序，由其调用（并以特权模式执行）模拟有问题指令的例程。这种机制与在 3.2.4 节中描述的调试工具完全一致（这也是在前面首先介绍这个工具的原因），并且可以推断出，它可能会干扰其他调试工具。正如本章参考文献［20］指出的，这种潜在的干扰需要将原始代码存储到位。这种技术避免了操作系统二进制代码的预处理。

采用半虚拟化，预处理是必要的。预处理使用所谓的超级调用替换所有特权指令，其中所提到的超级调用即虚拟机管理程序的系统调用。实际上，半虚拟化还涉及二进制重写，只是它的执行在前（而不是实时的）并且在整个操作系统的二进制代码上执行（而不是指令流的选择子集）。

这为虚拟机提供了一个软件接口，该接口与实际硬件 CPU 的接口不同。修改的客户机操作系统始终在环 1 中运行。需要注意的是，应用程序代码保持不变。半虚拟化的另一个有利特征是，即使使用完全的硬件辅助虚拟化，它也可以正常工作（尽管比需要的要慢），所以虚拟机也可以轻松地从不支持完全虚拟化的 CPU 迁移到一个支持的，而无须做任何更改（尽管如此，在速度上还是存在一定的损失：在半虚拟化的系统中，系统调用的目的是操作系统内核，首先在虚拟机管理程序中，然后需要与内核进行反复的交互）。

半虚拟化还有一些其他有趣的优势。值得注意的一个是计时器处理。所有现代操作系统都依赖于时钟中断来维护它们的内部计时器，这对于实时媒体处理特别重要。为此，即使空闲的虚拟机也需要处理时钟中断。通过半虚拟化，虚拟机代码被更改为在指定的时间请求通知。没有它，虚拟机管理程序将需要为空闲的机器安排计时器中断，根据本章参考文献［19］的作者指出的内容，这是不可扩展的。

另一个优点是可以采用多处理器架构工作。直到现在，假设在简化的 CPU 架构中只有一个 CPU，但这并不符合现代机器的情况。从理论上讲，操作系统处理多个 CPU 的方式与处理一个 CPU 的方式相同。通过模块化设计，只有调度程序和中断处理程序需要充分意识到两者之间的差异。这里没必要对此进行深入的探究，我们注意到，基于 x86 的多处理器架构采用用于中断重定向的高级可编程中断控制器（Advanced Programmable Interrupt Controller，APIC）来支持对称式多重处理（Symmetric Multi - Processing，SMP）。在虚拟模式下访问 APIC 的代价是高昂的，因为需要多次进入和退出虚拟机管理程序（参见本章参考文献［19］的代码例子）。借助可以查看全部代码的半虚拟化，多个 APIC 的访问请求可由一个超级调用来替代。

在处理 I/O 设备时，也可以实现类似的效果。总体来说，读者可能已经注意到了，半虚拟化实际上是一种编译方式，因此它拥有编译器享有的所有代码优化优势。

不幸的是，半虚拟化也有一些缺点。本章参考文献［19］的作者注意到，在将 x86 - 64 Linux 移植到 Xen 时，他们“明显体验到了纯软件半虚拟化的复杂性”，并认为导致这种复杂性的根本原因是强制“内核开发人员处理本就存在明显限制和与本地 CPU 相比具有不同行为的虚拟 CPU”。除此之外，这样的“行为”还需要新的中断和异常机制㊀，以及新的保护机制。

有趣的是，在某些情况下，如应用程序是 I/O 密集型的或需要大量的虚拟内存，与完全虚拟化相比，半虚拟化产生的执行速度更快。这一观察结果促使了混合虚拟化的发展。该想法不是通过修改半虚拟化接口来添加新功能，而是通过引入“伪硬件”，这是一个看似（对内核来说）硬件模块的内核接口。因此，半虚拟化接口本身保持最小化。创建新的“伪硬件”接口可用于内存管理（特别是 MMU）和所有的 I/O 设备。

接下来，将更加详细地对 I/O 虚拟化进行讨论。

3.3.4 I/O 虚拟化 ★★★

与我们喜爱的小狗一样，I/O 设备已经发展到能够识别它的主人——物理机器上的单个操作系统。事实证明，让它们对多个拥有者进行响应是难以实现的。正如本章参考文献［22］中指出的，“大多数的硬件本身不支持被多个操作系统访问”。

人们将更多的注意力集中在 CPU 虚拟化的本身，然而在虚拟环境下对 I/O 的处理却没有得到足够的重视。假设虚拟机管理程序本身应该能够处理执行实际的 I/O，同时使虚拟机“相信”它们拥有这些设备。

在某种程度上，I/O 虚拟化已经存在了很长一段时间，自从那时起操作系统就引入了假脱机的做法，作为与缓慢设备（例如，打印机、绘图机或现在已经淘汰的打卡机）的接口［最初，假脱机 SPOOL 是一个名词，它是外围设备同时联机操作（Simultaneous Peripheral Operations On - Line）的首字母缩写，但是到了 20 世纪 70 年代末，它已经成为了一个动词］。为了实现假脱机，操作系统为进程提供了一种看似像到设备的快速接口，而实际上，原始请求通过正确的格式化，同所有其他指令一起被存储在磁盘上，供系统异步处理。此外，操作系统为进程提供了针对每种设备类型（磁盘、磁带、终端等）统一的设备接口。每个设备的具体操作“隐藏”在驱动程序中。这样的设计支持逻辑设备和物理设备之间的解耦合，从而确保可移植性（更多详细信息请参见本章参考文献［23］）。

随着快速局域网的发展，I/O 虚拟化的另一个方面出现了。通过网络的方式，设备可以在网络上的机器之间共享。例如，对于网络上有一台机器带有打印机（有时甚至是磁盘）就足够了。该机器将通过网络接收 I/O 请求，代表发送方执行它们，并在需要时发送结果。通过使用互联网传输层协议［即传输控制协议（Transmission Control Protocol，TCP）或用户数据报协议（User Datagram Protocol，UDP），两者都在本书第 4 章中介绍］，可以实现这项功能。人们可以预见到不久的将来无磁盘云设备的实现，内存分页在网络上执行，其中只需要一台机器拥有真实磁盘即可。目前，业界已经开发了一种标准（IETF RFC 3720），可用于通过 TCP 传输小型计算机系统接口（Small Computer Systems Interface，SCSI）信息交换（SCSI 设计用于访问外围设备。这一开发在本书第 6 章有详细介绍，作为存储器讨论的一部分）。

㊀ 当内核地址空间需要共享时，这种保护机制问题就会出现。因为内核不再在环 0 中运行，所以无法使用环级保护，因此需要“在应用程序和内核之间的所有转换之间拼接地址空间的开销”[21]。

观察 I/O 虚拟化设计中的二元性是比较有趣的。由于几个虚拟机驻留在单个物理机器上，一方面需要在所有这些虚拟机之间共享一个 I/O 设备；另一方面需要在物理机器上模拟计算机网络的存在。稍后将讨论最后一点。现在，我们观察到，至少有必要在每个虚拟机上维护网络驱动程序。只要不让虚拟机直接访问设备，虚拟机管理程序则可以通过直接模拟设备或使用盘虚拟化驱动程序（有效地将设备驱动程序分为两个部分：虚拟机前端部分和虚拟机管理程序后端部分）来支持 I/O，如上一节所述。

实际上，某些级别的模拟也是必要的（并且一直存在于虚拟机管理程序中），即对低级固件[⊖]的模拟，如显卡和网卡或基本输入/输出系统（Basic Input/Output System，BIOS）。

BIOS 是一个很好的开始体现复杂性的例子。它通常在硬件芯片上实现，提供对（物理）机器硬件配置的访问。在正常情况下，BIOS 选择引导设备和引导系统。物理机器上只有一个 BIOS，因此虚拟机管理程序必须为每个虚拟机复制 BIOS 的功能。赋予所有的虚拟机完全访问硬件的能力所带来的潜在安全性问题不容低估，这使得它在虚拟机管理程序中的模拟成为一项不是很容易的任务。

在上述所有的选择中，采用直接访问 I/O 设备的选择是最快的，因此将其称为“直接 I/O”。为了理解其实现过程中所面临的问题，首先需要对图 3.2 中的计算机体系结构进行扩充。图 3.24 通过引入现在存在于每一个 CPU 中的直接内存存取（Direct Memory Access，DMA）设备扩展了这一体系结构。

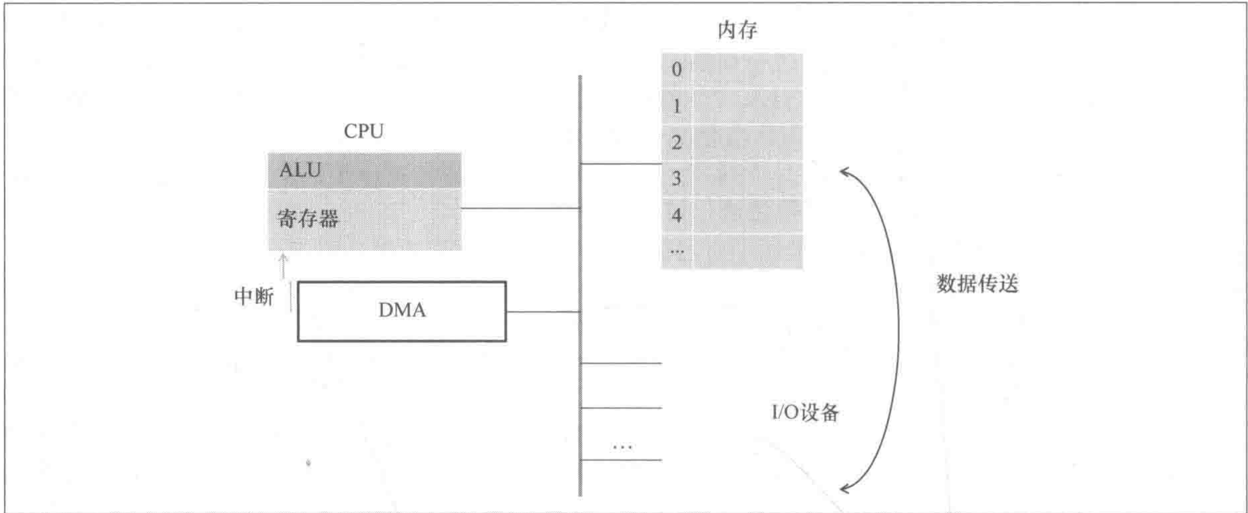


图 3.24 直接内存存取（DMA）

DMA 是用于将大量字节流传输到主存储器或从主存储器中传输出来的设备（或者确切地说，它是设备控制器的一部分）。需要这种需求的设备有磁盘和网络驱动器。如果让 CPU 亲自涉及这些 I/O 流的处理，那么 CPU 将没有足够的时间来进行它的主业——执行指令。DMA 可以承担这个任务，它的工作原理如下。DMA 可以访问设备和内存总线。当通过计算机时钟来驱动机器操作中的周期运转时，DMA 可以周期性地“窃取”一些周期代表 CPU 执行内存传输，或者，特别是在需要进行大量传输的情况下，DMA 在传输期间可以从 CPU 手中夺取并占用内存总线（在后一种情况下，CPU 将使用自己的高速缓存而不是主存储器进行工作，主存储器将在后期更新）。简而言之，DMA 被赋予内存和 CPU 之间的数据传输任务，并且在某些架构中甚至被用于内存内的数据传输。当 DMA 工作完成后，将中断 CPU。

DMA 的 I/O 虚拟化问题是指，在传统架构中 DMA 需要写入“真实”（即不是虚拟的）的内存。如果这种方法被严格遵循，直接 I/O 将为虚拟机提供未受保护的物理内存访问，而其中的内存是由多个虚拟机所共享的。所以，这样做将造成对安全性规则的公然违背。事实上，这也违背了隔离性原则，因为一个虚拟机中的恶意软件可能会危害到其他所有的虚拟机。

为了解决这个问题，人们采用了另一种类型的 MMU（称为 I/O MMU，有时也被拼写为 IOMMU），如图 3.25 所示。与图 3.11 中的 MMU 类似，其中 MMU 为 CPU 执行虚拟到实际的内存空间转换，I/O MMU 为 DMA 执行这一功能。因此，内存空间在设备之间被划分，并且转换和划分的高速缓存一般包含在硬件中，这通常提供内存保护（例如，Intel VT-D 方法，相关介绍内容请参见本章参考文献［21］）。

⊖ 固件是驱动硬件芯片的一种微型程序。有时，它是硬编码的（不可擦除），在有些情况下，也可以由用户进行修改。



这种方法支持虚拟机隔离并提供了一定程度的安全性。然而，如本章参考文献［21］中指出的，这里的问题是由动态转换映射引起的。这些问题与一般分页类似，总体上的问题是由虚拟机管理程序执行额外的工作导致的性能损失。实际上，这项工作需要在DMA完成内存传送之前执行，这又带来另一个DMA操作中容许延迟的问题。本章参考文献［24］中，对与使用I/O MMU相关的性能问题进行了分析。

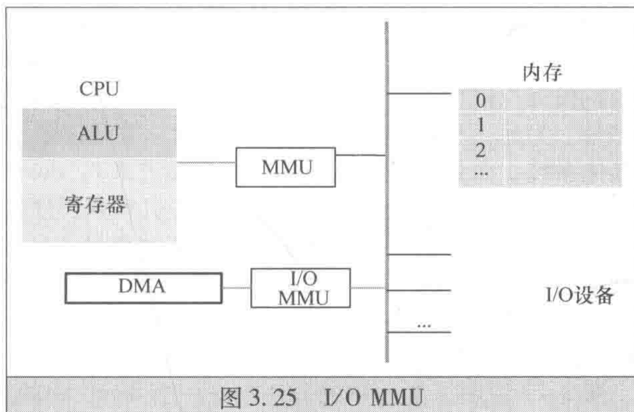


图3.25 I/O MMU

直接I/O以及整个I/O虚拟化方面仍然是一个热门的研究问题。知名著作“伯克利眼中的云计算”[25]在其论述的云部署的障碍类别中列举㊀“性能不可预知性”问题。I/O虚拟化取决于共享的I/O设备，这仍然是固有的依赖性问题（相反，共享主内存和CPU已经被证明可以有效地工作）。幸运的是，存在一些可以克服这个问题的想法，并且早期IBM大型机的成功也让人们在这些想法上看到了希望。

在这种乐观的情况下，我们结束了对虚拟化概念和技术的回顾。现在，可以用具体的主流虚拟机管理程序的例子来说明这些概念和技术的实现。

3.3.5 虚拟机管理程序实例 ★★★

在本节中，将通过几个广泛使用的虚拟机管理程序的实例，演示到目前为止所讨论的一些概念。这些虚拟机管理程序包括Xen虚拟机管理程序、KVM虚拟机管理程序以及VMware和Oracle开发的虚拟机管理程序。最后，将通过对NOVA微型虚拟机管理程序的介绍对本节内容进行总结，这将为后面有关虚拟机安全性部分内容的介绍提供一个自然的过渡。

从Xen虚拟机管理程序开始，本章参考文献［22］对其进行了全面的介绍。Xen是1型（“裸机”）虚拟机管理程序。它是操作系统不可知的，实际上它可以同时运行使用不同操作系统的客户机虚拟机。

就虚拟化分类而言，Xen支持半虚拟化和完全虚拟化客户机，分别称为PV和HVM（后者缩写代表“硬件辅助虚拟模式”。当然，HVM只能用在支持完全虚拟化扩展的处理器上，同时支持Intel和AMD的处理器扩展）。对于HVM中运行的客户机，Xen使用上一节描述的技术来模拟低级硬件盒固件部分，如显卡、网卡和BIOS适配器。可以预见，模拟通常会导致性能下降。Xen通过另一种称为PV-on-HVM（或PVHVM）的模式来处理这一问题，其中HVM客户机仅部分半虚拟化。

Xen处理I/O设备的方法简单而优雅。为了解释它，需要在图3.26的帮助下引入Xen域的概念。

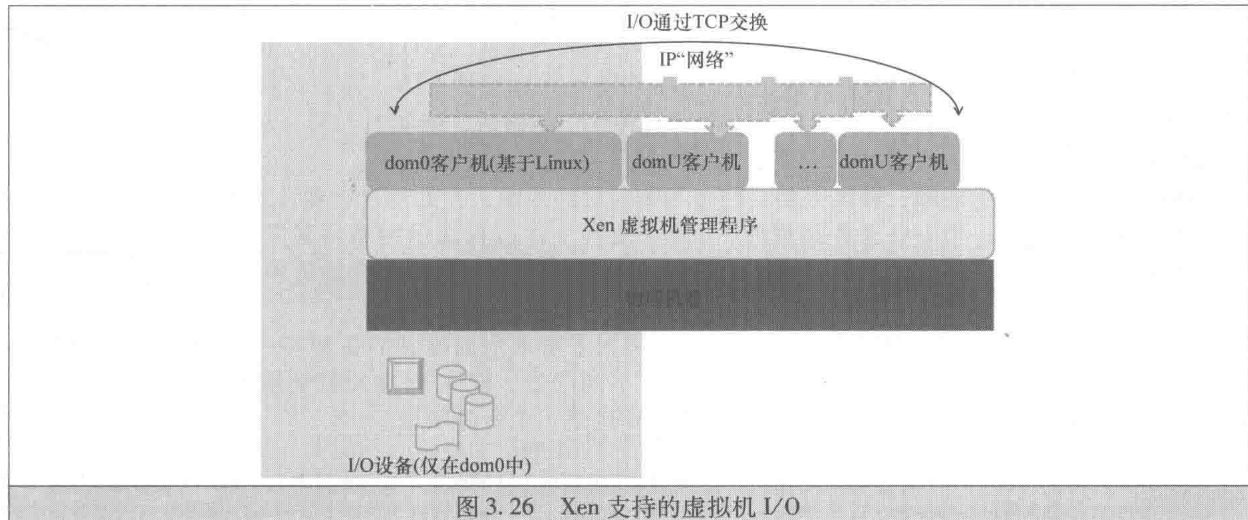


图3.26 Xen支持的虚拟机I/O

㊀ 该障碍编号为5。

Xen 为每个客户机创建了一个称为“域”的特殊环境。“正常”的客户机（除了一个特殊的客户机以外的所有其他客户机）都在它们自己未经授权的域 domU 中运行。域0（或 dom 0）作为一个特殊域，是为特权客户机预留的。特权客户机被称为“虚拟主机”，因为它是托管其他所有虚拟机的控制系统的一部分。为此，dom 0 在 Xen 启动时创建，并且仅限于运行专用的操作系统（一般是 Linux，但是也支持其他系统，而且不时还有新的系统添加进来）。dom 0 还是指定策略的地方。控制机制使用的机制和策略的分离是 Xen 的设计原则。Xen 虚拟机管理程序根据 dom 0 中的配置策略执行这些机制。

就 I/O 而言，这个诀窍就在于，所有物理设备只能附加到处理它们的 dom 0 客户机上，具有已分配的独特权限。这解决了 I/O 设备识别一个拥有者的问题。所有其他的客户机都明白，它们的机器没有连接 I/O 设备，但所有这些设备都位于 dom 0 客户机所在的机器上。这不是问题，因为所有的虚拟机都是网络可寻址的，因此可以使用网络 I/O。

图 3.27 说明了 Xen 中的 I/O 操作。当 domU 中的客户机需要与设备通信时，它将使用拆开的 I/O 驱动器的上半部分[⊖]。

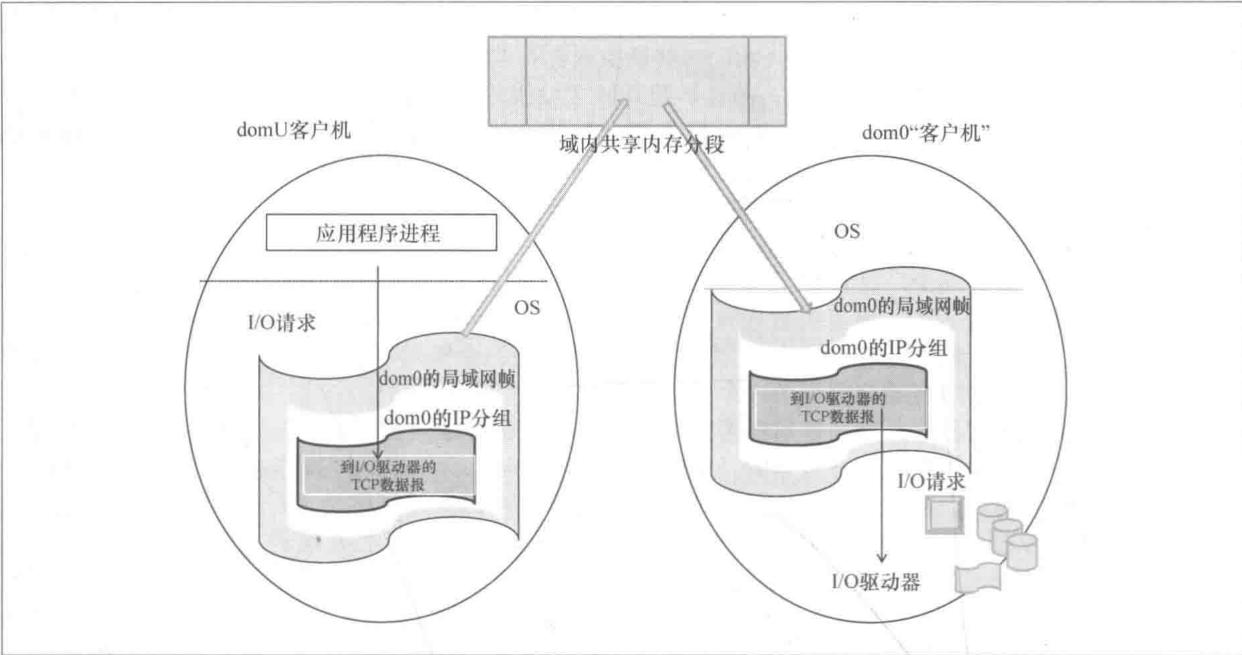


图 3.27 使用共享内存的 Xen 网络 I/O 优化

I/O 请求使用网络模块通过 TCP 被发送到 dom 0。在此过程中，构建了一个 TCP 数据报，并将其封装成 IP 分组。然后将 IP 分组封装到链路层帧内，作为对“真实”I/O 的请求，由网络驱动程序通过局域网（Local Area Network，LAN）发送出去。Xen 没有简单地模拟 LAN，而是将帧放置在物理机器上公共（对 domU 和 dom 0 都可见）的内存分段中（这种优化的做法，确实引入了一定的安全风险，因此需要谨慎的保护方案）。

dom 0 接收 LAN 上的帧，剥离链路、网络和传输层的封装，并处理发送到针对相应设备的实际 I/O 驱动程序的 I/O 请求。当 I/O 工作结束时，dom 0 反转上述过程通过网络将其发送回来。应该注意的是，实际的操作要比在此处的简单介绍复杂得多，因为它涉及支持 TCP 会话的多次交换，正如 TCP 所规定的会话过程。

KVM（首字母缩略词有时以小写字母拼写）利用了本章前面提到的新型客户机操作模式。和 Xen 一样，KVM 也是 1 型的虚拟机管理程序[⊜]；与 Xen 不同的是，KVM 将虚拟机创建成为一种能够运行在 Linux 内核上运行的 Linux 进程。同 Xen 一样，KVM 也是一个开源项目。

⊖ 拆分驱动程序是所有实现异步 I/O 处理机制的通用术语。在网络 I/O 情况下，驱动程序的上半部分负责对需要执行 I/O 处理的进程隐藏设备的位置（以及设备不在特定机器上的情况）。上半部分驱动程序负责将 I/O 请求发送到设备所在的机器上，当然也可以接收结果。下半部分驱动程序负责与实际的设备相关的各种操作。

⊜ 稍后将再次回到这种描述上来，因为在这种情况下，类型 1 和类型 2 之间的区别不是特别明晰。



KVM 可以在扩展的 x86 CPU（包括 Intel VT 和 AMD - V）以及其他的处理器上支持完全的本地虚拟化，并且这一列表仍在扩展。

除了完全的本地虚拟化以外，KVM 还以半虚拟化网络驱动程序的形式支持基于 Linux 和微软 Windows 操作系统的客户机的半虚拟化。在 Quick EMUlator（QEMU）的帮助下，KVM 的修正版可以支持 Mac OS X。

同样，借助 KVM 的架构[19]，可以使用与常规 Linux 进程相同的接口创建虚拟机。实际上，每个虚拟 CPU 对管理员来说似乎都是一个常规的 Linux 进程，尽管这主要是用户界面的问题。

KVM 使用一个 QEMU 的修改版本来提供设备模拟，该版本模拟 BIOS 以及扩展的总线扩展、磁盘控制器（包括那些 SCSI）、网卡和其他的硬件盒固件部分。

具体来说，通过打开 Linux 设备节点（/dev/kvm）创建虚拟机。作为一个新的进程，机器有自己的内存、与创建它的进程分开。除了创建一个新的虚拟机之外，/dev/kvm 还提供了对内存分配、机器的 CPU 寄存器访问以及中断 CPU 的控制。

使用 Linux 内存管理功能，内核可以在不连续的分页上配置客户机地址空间（如图 3.13 所示）。用户空间也可以映射到物理内存中，这有助于模拟 DMA。

KVM 通过引入新的执行模式（访客模式）来修改 Linux，转换图如图 3.28 所示（与 UNIX 操作系统类似，Linux 支持其他两种模式，即内核模式和用户模式）。

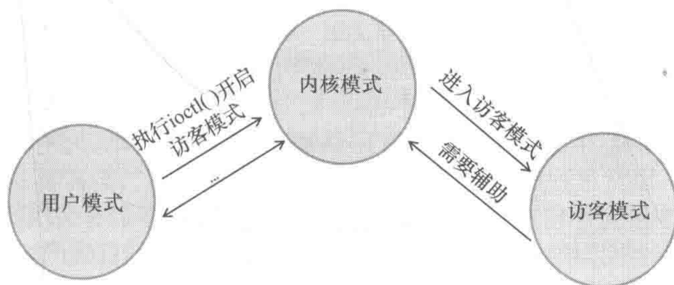


图 3.28 KVM 模式状态转换图

其工作流程如下，用户级代码通过 Linux 的 ioctl（ ）函数发出系统调用，请求执行访客代码。作为响应，内核开启访客代码在访客模式下的执行。这将持续到退出事件（例如，I/O 指令请求或中断或超时）的发生。当 CPU 退出访客模式时，它将返回到内核模式。内核处理事件，并根据其结果来判断是恢复访客模式，还是返回到用户模式。

正如本章参考文献［19］所述，与 Linux 的高度整合有助于 KVM 的开发人员和用户改造很多 Linux 功能（例如，调度、进程管理和特定 CPU 的虚拟机分配）。

之前说 KVM 是 1 型虚拟机管理程序，但开发者社区对此存在不同的看法。有些人认为 KVM 是通过 Linux 接口来管理的，因此事实上，虚拟机是作为 Linux 进程出现的，所以它是 2 型虚拟机管理程序。这里的分界线是通过虚拟机管理程序运行所在的权限环作为标准来划定的，而恰巧，KVM 代表的 Linux 子系统具有最高的权限（更不用说它运行在裸机上，但是所有的代码也是如此，除非它由软件解释。在我们看来，该定义的区别在于授予虚拟机管理程序代码的权限）。

KVM 作为 1 型虚拟机管理程序的问题受到质疑，而纯粹的 2 型虚拟机管理程序有 VMware® Workstation 和 Oracle VM VirtualBox®㊀（相比之下，VMware ESX 和 ESXi 虚拟机管理程序是 1 型的）。

在回顾 x86 虚拟化中引入了本章参考文献［18］的这篇论文，还提供了对 VMware Workstation 虚拟机管理程序的全面介绍（这篇论文的作者更喜欢用 VMM 这个词）。与 Xen 不同，VMware Workstation 通过扫描二进制映像并使用陷阱或直接库过程调用替换有问题的指令，来近乎实时地处理未捕获的特权指令问题（需要记住的是，这里的区别在于 Xen 使用半虚拟化，通过对操作系统代码的预处理来实现这种类似的结果；VMware 不需要这样的预处理，映像加载后，代码将随即更改）。

另一个 2 型虚拟机管理程序，Oracle VM VirtualBox（其用户手册[26]给出了全面的描述）采用了与

㊀ VirtualBox 软件最初由 Innotek GmbH 公司开发，该公司后来被 Sun Microsystems 收购，后者后来又被 Oracle 公司收购。

半虚拟化和完全软件模拟不同的技术组合（尽管在特殊情况下使用后者）。

VirtualBox 软件通过其环 0 内核驱动程序建立主机系统，使访客代码实现本地运行，VirtualBox 潜伏在“下面”，随时可以在需要的情况下进行控制。用户代码（环 3）未修改，但 VirtualBox 重新配置访客虚拟机，使其系统（环 0）代码能够运行在环 1 中。当系统代码（现在在环 1 中运行）尝试执行特权指令时，VirtualBox 虚拟机管理程序会拦截控制。在这一点上，可以向主机 OS 提供指令执行，也可以通过 QEMU 重新编译器运行指令。

当 VirtualBox 使用自己的反汇编器㊀调用重新编译来分析代码时，本章参考文献［26］中提到的一个有趣的例子是“当访客代码禁用中断时，VirtualBox 无法确定这些中断何时会被重新启动”。所有的保护模式代码（如 BIOS 代码）也被重新编译。

重要的是，应当注意到，目前提到的所有虚拟机管理程序都支持实时迁移。这个问题很重要，因为实时迁移涉及虚拟机内存的复制，而这个内存仍然可以被修改（在所有内存复制完毕之前，机器自然需要在旧主机上继续运行），所以它是一个多重传递过程，需要在旧主机和新主机之间精确地进行同步。除了内存转移，还有其他方面实时迁移的问题，稍后将在本书中讨论这些问题。

与此同时，人们为了解决操作系统安全性的问题，通过让操作系统内核尽可能地小而简单，正在开发新的虚拟机管理程序（如 NOVA[27]）（通过简单的量化对比可以发现，Xen 虚拟机管理程序有约 15 万行代码，而 NOVA 虚拟机管理程序只有 9000 行代码）（本章参考文献［27］提供了一个表格，对本节提到的大部分虚拟机管理程序的代码量进行了对比）。

虽然 NOVA 并不否定半虚拟化，但本章参考文献［28］的作者称：“我们并没有在我们的系统中使用半虚拟化，因为我们既不想依赖于客户机操作系统源代码的可用性，也不需要将操作系统移植到半虚拟化接口的额外工作”。NOVA 也不使用二进制翻译，它只依赖于完全虚拟化的硬件支持。

NOVA 代表了第三代的微内核（或 μ-内核）系统，这些操作系统只能为应用程序提供最基本的功能：内存和进程管理，以及进程间通信。这种方法背后的理念是，当需要执行敏感操作时，其余的操作系统功能（包括文件系统管理），可以由陷入到内核的应用程序本身来执行。微内核最初是与 LAN 结合开发的，但后来的安全性问题（它需要软件上的简单性）成为其发展的主要驱动力。

NOVA 微虚拟机管理程序是采用微内核作为内核的虚拟机管理程序。它仅提供最基本的虚拟化、虚拟机分离、调度、通信和平台资源管理机制，其余的功能都交由上层提供。为此，每个虚拟机均有其自己关联的虚拟机监视器，作为无特权的用户应用程序在微虚拟机之上运行。这种架构如图 3.29 所示。

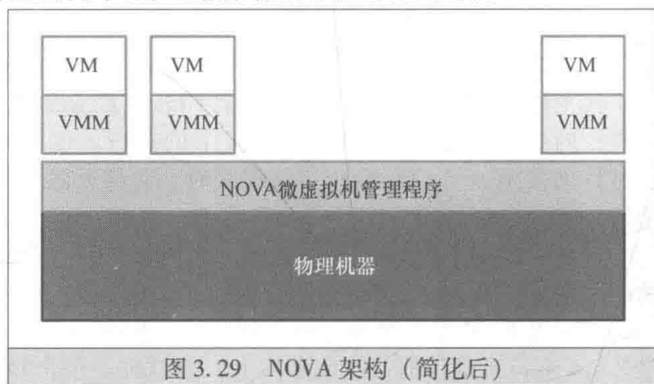


图 3.29　NOVA 架构（简化后）

需要注意的是，这种架构与迄今为止所回顾的所有其他架构截然不同，因为它的虚拟机管理程序分为一个整体的微内核部分和为每个虚拟机复制的用户部分。这里最重要的是，微考内核的设计专门虑了安全性：虚拟机管理程序的分解使特权代码的数量实现了最小化。

但是，正如本章参考文献［28］的作者指出的：“通过在用户级实现虚拟化，改善了安全性，但在性能方面略有下降”。

通过上述讨论可以引出本章下一节（也是最后一节）的主题，即安全性。

3.3.6　安全性 ★★★

安全是迄今为止云计算中最复杂的问题。云计算结合了多种技术，所有与这些技术相对应的每一种威胁也得到了组合或合并。同时，不同服务和部署模式下的安全需求，又使这个问题变得进一步的复杂化。这需要在云计算中采取三管齐下的安全措施：①了解每种技术（操作系统、虚拟化、路由和交换层的数据网络、网络应用程序，这里仅列出主要的技术）所带来的安全威胁（并评估其风险）以

㊀ 反汇编器是一种能够将二进制代码转换成汇编级代码的程序。在这种情况下，可将其用于分析应用中。



及各自的缓解机制；②根据具体的服务模式和特定的部署模式分析风险和机制；③制定整体的安全图示。本书将在一章内系统地处理通用的安全机制，而属于各章特定的技术将分别在各章内进行介绍。因此，本章仅讨论虚拟化的安全问题，其最终被归结为虚拟机管理程序的安全问题。

关注虚拟机管理程序安全问题的动机现在看来应该是显而易见的。虚拟机管理程序是一个单独的安全故障点。如果它受到威胁，所有受它控制的虚拟机也都将受到威胁。那么，它是如何受到威胁的呢？例如，某种精心设计的蠕虫（在3.2.3节中讨论的）可以“逃离”用户进程的代码并接管只由操作系统才可以控制的计算机资源。因此，攻击中的恶意访客代码（实际上被称为虚拟机逃离攻击[27]）可以突破虚拟机，控制虚拟机管理程序，从而控制同一物理主机上的所有其他虚拟机。

假设当前主机硬件已经被完全保护，防止篡改，并且引导过程是安全的，这些攻击可能以恶意修改操作系统和设备驱动程序的形式出现，调度DMA在没有I/O MMU的情况下写入“错误的”内存。虚拟机与主机之间接口细节之处的问题，用本章参考文献［27］的话来说，它提供了“一个攻击面，恶意客户机操作系统可以利用它来攻击虚拟化层”。如果接口中存在漏洞（并且众所周知，没有不存在漏洞的软件规范），则可以利用它来使虚拟机接管虚拟机管理程序，或者以相同的权限级别来执行（这就是NOVA架构为每个虚拟机提供专用VMM的原因，这个功能增加了一个额外的隔离层。即使虚拟机接管它的作为无特权应用程序运行的VMM，也不会对顶层的虚拟机管理程序造成损害）。

当引导程序遭到侵害时（通常通过对未受保护的固件的攻击），可能会导致安装的恶意程序获得最高的运行权限，即在UNIX系统中获得root权限，人们通常将这种类型的程序称为rootkit。最糟糕的是，rootkit一般无法被检测到。

本章参考文献［29］简要介绍了有关rootkit的历史。rootkit最开始以“特洛伊木马㊀”的形式出现，此时它主要是以用户级别的权限运行，运行之后将自己隐藏起来，危害性较小。但是，随着管理工具的出现，这些工具能够检测到它们，因此rootkit逐渐发展成为内核级的恶意程序。只要入侵检测工具能够发现它们，它们就会找到新的方式躲避检测。

rootkit或者具有类似rootkit隐身表现的非恶意程序，已经被人们用于看似合法的目的（执行版权保护就是这样的一个例子），不过在法律方面这似乎是一个灰色地带。人们对rootkit进行了大量的学术研究，旨在发现rootkit可能利用的漏洞。本章参考文献［29］中描述的SubVirt项目，由微软和密歇根大学共同开发。基于虚拟机的rootkit（Virtual Machine - Based Rootkit，VMBR）是该项目的一项成果。VMBR通过操纵系统启动顺序将自身插入到虚拟机管理程序和目标操作系统之间。这既可以通过远程漏洞（本章参考文献［29］中未具体指出什么漏洞）实现，也可以通过社会工程（欺骗用户、买通供应商，或破坏“对等网络上存在的可引导的CD - ROM或DVD镜像”）的方法实现。一旦获得了root权限，VMBR就会被安装在磁盘上（这需要对一些磁盘区块及其包含的数据进行窜改）。此时，引导记录也以避免这种修改被实际检测到的方式被巧妙地修改了。为了实现这一点，“在大多数进程和内核子系统退出之后，在关机的最后阶段”，引导区块被操纵。据报道，微软的Windows XP和Linux操作系统都被这种方式入侵过，本章参考文献［29］对相关技术细节进行了介绍。就虚拟机管理程序而言，据报道，已经在Linux/VMware和微软Windows/VirtualPC平台上实现了概念性验证VMBR。

也许VMBR仍然是一个鲜为人知的学术活动，研究人员没有将它投入到实际的运用情况（在这种情况下，可以进行一系列的“服务”，例如，“键盘敲击嗅探器、钓鱼式攻击网络服务器、用于搜索用户敏感数据文件的工具，以及防止普通VMM检测技术的检测对策”）。研究人员还演示了，他们的VMBR可以有效地修改操作系统中可以观察到的状态，这使得它非常难以被发现——确实很难发现，以至于本书的作者之一使用一台已经被注入我们的概念性验证VMBR的机器，却完全没有意识到。

检测这类rootkit的防御策略的关键点是在VMBR以下的层上运行检测软件，以便它能够“读取物理内存或磁盘并查找指示VMBR存在的签名或异常，例如，被修改了的启动顺序”。这种级别的保护可以在固件中实现。还建议使用安全引导技术进行预防，在我们看来，这是最好的方法。但是，如果通过社会工程的方法，让安全的启动介质（即压缩的CD - ROM）中毒，则上述方法将失去效果。

SubVirt VMBR是针对标准情况（不可虚拟化的x86 CPU）实施的，而另一个研究项目开发的Blue Pill VMBR则是专为完全可虚拟化的AMD芯片设计的。Blue Pill的代码对外免费提供，但是人们对这个

㊀ 特洛伊木马是一种看起来像（或声称自己是）另一种（一般是知名的）应用程序的程序。

项目存在重大的争议，因此很多公司阻止通过它们的公司网络访问这些网页。

到目前为止，本书已经分析了一些安全方面的情况，包括恶意软件“接管”虚拟机管理程序，或者以虚拟机管理程序级别的权限运行。事实证明，控制虚拟机管理程序本身，可能不会造成什么损害。但利用虚拟机隔离中的漏洞，则足以造成损害。

即便是一个不经意（被动的）攻击，只要攻击者知道哪些是秘密信息，也会造成严重的损害。为此，重要的是要了解并记住与物理计算机内部工作有关的信息，即便这些信息看起来没什么重要性，也会被泄露并造成重大损害后果。存在一系列的旁道攻击（Side - Channel Attack），其中硬件（例如，CPU、内存或其他设备）使用模式在攻击者利用下可以泄露出各种秘密信息，包括加密密钥。

承载虚拟机的物理主机的相关信息必须给予保密（后面将会提到，这与网络和服务提供商在保护基础设施相关信息安全方面长久以来的做法非常相似）。这就解释了本章参考文献［30］㊀的作者提供的实验报告［他们的实验使用了大型云提供商针对 Xen 虚拟机管理程序（并基于 dom 0 IP 地址的比较）推出的服务］会在业界中引起巨大轰动的原因。

研究人员通过演示证明，“获得内部云基础设施的映射图，确定特定目标 VM 可能的驻留位置，然后实例化新的 VM 与目标 VM 共同驻留是可能的”。一旦实例化，机器就会开启跨虚拟机的旁道攻击，从而得到（通过 CPU 数据缓存）CPU 的利用率测量值。基于这些数据可以进行各种攻击，包括键盘敲击计时（其中攻击者可以利用两次连续敲击键盘的时间间隔猜测输入的密码），本章参考文献［30］介绍了有关这方面内容。

在推荐的各种降低风险的方法中，本章参考文献［30］的作者提到了云提供商服务的内部结构和置放策略之间的问题。例如，提供商可以禁止基于网络的共同驻留检测。还有一种有点模糊的建议是“采用盲法技术最大限度地减少可能泄露的信息”。但是，问题在于人们无法预测什么类型的信息可能会被攻击者用作旁道。本章参考文献［30］的作者认为，最好的解决方案是“将风险和置放决策直接暴露给用户”。因此，用户可能需要专用的物理机器并为此承担相应的费用。不幸的是，这就违反了云计算的原则。目前，将旁道攻击作为上述方案的理由在人们看来还有些牵强。但是，保护云提供商基础设施信息的意识却是重要的、合理的，也是及时的。在本书后面的内容中，将讨论实现这种做法的机制。

在本章的内容中，介绍了与虚拟机管理程序相关的一些 NIST 建议[27]：

1）供应商发布虚拟机管理程序之后，要及时安装与之相关的所有更新。大多数的虚拟机管理程序都有自动检查更新的功能，并在发现更新的时候就会自动安装。集中式的补丁管理方法也可以用于管理更新。

2）限制对虚拟机管理程序接口的管理访问。使用专用管理网络保护所有管理通信渠道，或者使用 FIPS 140 - 2 验证的加密模块㊁对管理网络通信进行验证和加密。

3）将虚拟化基础设施同步到受信任的权威时间服务器。

4）从主机系统断开未使用的物理硬件。例如，可插拔磁盘驱动器可能偶尔用于备份或还原，但是当它没有被使用时，应当将其与主机的连接断开。断开所有网络上未使用的 NIC㊂（网卡）。

5）禁止所有的虚拟机管理程序服务，如客户机操作系统和主机操作系统之间的剪贴板或文件共享，除非需要用到它们。这些服务中的任何一个都可以成为攻击者利用的工具。在多个客户机操作系统与主机操作系统共享同一个文件夹的情况下，文件共享也可以成为这种情况下系统上的一种攻击媒介。

6）考虑使用自省功能㊃来监控每个客户机操作系统的安全性。如果客户机操作系统遭到威胁，则其安全控制可能会被禁用或重新配置，以便隐藏任何受到威胁的迹象。在虚拟机管理程序中内置安全服务可以实施安全监控，即使在客户机操作系统受到威胁时。

㊀ 强烈建议读者完整地阅读这篇论文，同时建议不熟悉 IP 网络的读者先来学习第 4 章内容，再回过来阅读这篇论文。本书第 4 章中介绍了一些必备的网络背景。

㊁ 将在本书后面内容中详细讨论这一问题。

㊂ NIC 代表网络接口控制器，它是一种将计算机连接到 LAN（在物理层）的设备。

㊃ “自省功能”这一术语指的是那些超出正常虚拟机管理功能的虚拟机管理程序的功能。这些功能包括监控虚拟机之间的业务流量、访问文件系统、内存和程序执行，以及所有有助于检测入侵的其他活动。

7）考虑使用自省功能来监控客户机操作系统之间产生的活动的安全性。这对于在非虚拟化环境中通过网络进行传输并通过网络安全控制［例如，网络防火墙、安全设备与网络入侵检测和防护解决方案（Intrusion Detection and Prevention Solution，IDPS）传感器）进行监控的通信来说尤其重要。

8）仔细监控虚拟机管理程序本身是否有遭到威胁的迹象。这包括使用虚拟机管理程序提供的自我完整性监控功能，以及对虚拟机管理程序日志进行持续的监控和分析。

顺便说一句，业界中存在一种观点，只要代码量大小增加，就会增加被攻击的可能性，因此认为带有自省服务的虚拟机管理程序可能会提供额外的攻击媒介。同时，这样做在性能方面也存在一定的问题。

另一种方法是开发一种特权设备，即通过对外开放虚拟机管理程序的 API 来提供自省服务的虚拟机。这样既考虑了性能问题（因为这种服务是在设备层面上执行的），又解决了虚拟机管理程序代码蠕变的问题。但是，如果没有对自省 API 的访问加以控制以将其仅限定在特定设备上，则对外开放这种 API 可能是非常危险的。

然而，还有一种工作在受控环境（如私有云）中的自省方法，即完全避开虚拟机管理程序，而是在每个虚拟机中安装类似于 rootkit 的代理。由这些代理来监控虚拟机的行为并将它们记录下来。

人们一直在进行着有关虚拟化安全性方面的研究。普林斯顿大学正在研究一种名为 NoHype 的新方法[31, 32]。在这种方法中，虚拟机管理程序只是为虚拟机准备运行的环境，然后将其分配给仅由相应机器“拥有的”特定的 CPU 核心。这种方法可以适用于特别敏感的情况，其中使用了最高级别的虚拟机隔离（几乎使用了独立的计算机）。

参考文献

[1] Creasy, R. J. (1981) The origin of the VM/370 time – sharing system. IBM Journal of Research and Development, 25 (5), 483 – 490.

[2] Tanenbaum, A. S. (2006) Structured Computer Organization, 5th edn. Pearson Prentice Hall, Upper Saddle River, NJ.

[3] Knuth, D. (1997) Art of Computer Programming, Volume 1: Fundamental Algorithms, 3rd edn. Addison – Wesley Professional, New York.

[4] Eichin, M. W. and Bochlis, J. A. (1989) With microscope and tweezers: An analysis of the Internet virus of November 1988. Proceedings of the 1989 IEEE Symposium on Security and Privacy, May 1 – 3, Cambridge, MA, pp. 326 – 343.

[5] Stravinsky, I. (1988) Petrushka, First Tableau (full score). Dover Publications, Mineola, NY, p. 43.

[6] Tanenbaum, A. S. (2007) Modern Operating Systems, 3rd edn. Prentice Hall, Englewood Cliffs, NJ.

[7] Silberschatz, A., Galvin, P. B., and Gagne, G. (2009) Operating System Concepts, 8th edn. John Wiley & Sons, New York.

[8] Bach, M. J. (1986) The Design of the Unix Operating System. Prentice Hall, Englewood Cliffs, NJ.

[9] Motorola (1992) Motorola M68000 Family Programmer's Reference Manual. Motorola, Schaumburg, IL.

[10] Organick, E. I. (1972) The Multics System: An Examination of Its Structure. MIT Press, Boston, MA.

[11] Graham, R. M. (1968) Protection in an information processing utility. Communications of the ACM, 11 (5), 365 – 369.

[12] Kidder, T. (1981) The Soul of a New Machine. Atlantic, Little, Brown, Boston, MA.

[13] Goldberg, R. P. (1973) Architecture of Virtual Machines, 2nd edn. Proceedings of the AFIPS National Computer Conference, June 4 – 8, Montvale, NJ, pp. 74 – 112. http://flint.cs.yale.edu/cs428/doc/goldberg.pdf.

[14] Popek, G. J. and Goldberg, R. P. (1974) Formal requirements for virtualizable third generation architectures. Communications of the ACM, 17 (7), 412 – 421.

[15] Robin, J. S. and Irvine, C. E. (2000) Analysis of the Intel Pentium's ability to support a secure Virtual Machine Monitor. Proceedings of the 9th USENIX Security Symposium, August 14 – 17, Denver, CO, pp. 129 – 144.

[16] Intel Corporation (2010) Intel® 64 and IA – 32 Architectures, Software Developer's Manual, Volume 3A: System Programming Guide, Part 1. Order Number: 253668 – 034US, March.

[17] Advanced Micro – Devices (2012) AMD64 Technology. AMD64 Architecture Programmer's Manual, Volume 3: General – Purpose and System Instructions. Publication No. 24594 3.19. Release 3.19, September.

[18] Adams, K. and Agesen, O. (2006) A comparison of software and hardware techniques for x86 virtualization. ASPLOS – XII Proceedings of 12th International Conference on Architectural Support for Programming Languages and Operating Systems, New York, pp. 412 – 421.

[19] Kivity, A., Kamay, Y., Laor, D., et al. (2007) kvm: The Linux Virtual Machine Monitor. Proceedings of the Linux Symposium, June 27 - 30, Ottawa, Ont., pp. 225 - 230.

[20] Nakajima, J. and Mallick, A. K. (2007) Hybrid - virtualization—enhanced virtualization for Linux. Proceedings of the Linux Symposium, June 27 - 30, Ottawa, Ont., pp. 86 - 97.

[21] Intel (2006) Intel® virtualization technology for directed I/O. Intel® Technology Journal, 10 (03). ISSN 1535 - 864x, August 10, 2006, Section 7.

[22] Chisnall, D. (2007) The Definitive Guide to the Xen Xypervisor. Prentice Hall Open Source Development Series, Pearson Education, Boston, MA.

[23] Waldspurger, C. and Rosenblum, M. (2012) I/O virtualization. Communications of the ACM, 55 (1), 66 - 72.

[24] Ben - Yehuda, M., Xenidis, J., Ostrowski, M., et al. (2007) The price of safety: Evaluating IOMMU performance. Proceedings of the Linux Symposium, June 27 - 30, 2007, Ottawa, Ont., pp. 225 - 230.

[25] Armbrust, M., Fox, A., Griffith, R., et al. (2009) Above the Clouds: A Berkeley view of Cloud computing. Electrical Engineering and Computer Sciences Technical Report No. UCB/EECS - 2009 - 2A, University of California at Berkeley, Berkeley, CA, February 10.

[26] Oracle Corporation (n. d.) Oracle VM VirtualBox® User Manual. www. virtualbox. org/manual/.

[27] Scarfone, K., Souppaya, M., and Hoffman, P. (2011) Guide to Security for Full Virtualization Technologies. Special Publication 800 - 125, National Institute of Standards and Technology, US Department of Commerce, January.

[28] Steinberg, U. and Kauer, B. (2010) NOVA: A microhypervisor - based secure virtualization architecture. Proceedings of the 5th European Conference on Computer Systems (EuroSys'10), Paris, pp. 209 - 222.

[29] King, S. T.; Chen, P. M., Wang, Y., et al. (2006). SubVirt: Implementing malware with virtual machines. Proceedings of the 2006 IEEE Symposium on Security and Privacy (S&P'06), Oakland, CA, pp. 327 - 341.

[30] Ristenpart, T., Tromer, E., Shacham, H., and Savage, S. (2009) Hey, you, get off of my Cloud: Exploring information leakage in third - party compute Clouds. Proceedings of CCS'09, November 9 - 13, 2009, Chicago, IL.

[31] Keller, E., Szefer, J., Rexford, J., and Lee, R. B. (2010) NoHype: Virtualized Cloud infrastructure without the virtualization. ACM SIGARCH Computer Architecture News, 38 (3), 350 - 361.

[32] Szefer, J., Keller, E., Lee, R. B., and Rexford, J. (2011) Eliminating the hypervisor attack surface for a more secure Cloud. Proceedings of the 18th ACM Conference on Computer and Communications Security, ACM, pp. 401 - 412.

第4章 数据网络——云的神经系统

本章所述的内容在第3章中已经提到过。在第3章中，不止一次地提到了数据网络，因此，本章将对这一问题进行全面的介绍和深入的研究。

数据网络指的是能够实现计算机与计算机通信的一组技术。最终带来的结果是，两个位于不同计算机上的进程可以实现彼此通信。反过来，这又可以支持分布式处理。读者可能记得，在同一个机器上实现进程间通信是操作系统的任务。因此，通过网络在机器之间实现这种功能自然也会涉及操作系统。

事实上，数据通信和操作系统的学科自20世纪60年代以来一直在发展。在早期的系统中，通过用于物理网络访问的新设备驱动程序和用于跨网络的进程间通信的专用库（与机器中内核提供的进程间通信库不同）这两种方式来添加数据通信能力。有趣的是，早期的跨网络进程间通信本身并不是目的[㊀]，它只是一种用来实现对远程资源（通常是文件）访问的手段。文件传输是主要的数据网络应用。另一个是交易系统（例如，在银行或航空公司票务预定中使用的那些系统），其中用户输入一条命令，可以引起远程机器数据库做出相应的动作，从而从这台机器获得响应。

到20世纪80年代末，操作系统的演进沿着两个分支方向进行：①网络操作系统，它提供了一种环境，用户可以访问远程资源，并意识到这些资源是远程的；②分布式操作系统，它提供了一种环境，用户可以像访问本地资源一样访问远程资源。这一发展本身就非常有趣，其结果在本章参考文献［1］中有很好的描述。此外，我们推荐本章参考文献［2］作为一本有关分布式计算的百科全书式参考书籍。不止一种方式，这些发展目标与云计算的目标非常相似，因为这些目标包括对迁移的支持，体现在3个方面：数据迁移涉及将文件或其中的一部分移动到访问机器；计算迁移涉及在远程计算机上调用所需的计算（计算所需的数据驻留在远程机器上），并将计算结果返回给访问机器；进程迁移涉及在远程机器上执行程序（例如，出于负载平衡的目的）。当然，云计算可以通过完全虚拟机（而不是进程）迁移的方式来加强自身的功能。不过，一种新兴的趋势是在廉价的微处理器上运行新的应用程序。这样，CPU虚拟化部分在弹性实现方面就变得不那么重要，而数据网络的重要性仍将突出。

由于篇幅有限，本书无法专注于分布式处理的细节内容。幸运的是，在这方面有很多著名的专著（包括刚刚提及的参考文献）。在本章余下的内容中，将主要在万维网方面介绍有关分布式计算的内容。

回到数据网络，至少它需要计算机的物理互连，如图4.1a所示。当然，这种形式的互连是云计算的一个重要方面，因为它会影响云中、任意两个联合云之间，以及需要访问云端的计算机与云之间的数据交换。

值得注意的是，图4.1b体现的双重性（在上一章中已经不止一次提到过）。这里，因为物理机器“转换”为虚拟机整合在一台物理机器上，所有网络软件都必须保持不变，所以现在位于同一台物理机器中的“物理网络”实际上由虚拟机管理程序进行有效的模拟。

就数据网络内容的介绍而言，尽量让本书的内容自成体系且内容完整。

从数据网络的概述开始，使用国际标准化组织（International Organization for Standardization，ISO）制定的经典开放系统互连（Open Systems Interconnection，OSI）模型。OSI参考模型是解释数据通信原

㊀ 同语音和文本通信一样，那时人们也考虑了通过这种方式进行资源通信。

理机器主要问题的最佳通用工具。所有的数据通信协议都（或多或少地）适合该模型。并且，特定协议适用的问题通常会使人们对所涉及问题产生更深入的理解。

在讨论了该模型之后，介绍了网际协议（Internet Protocol，IP），并回顾了互联网协议族。在回顾的过程中，很显然，尽管互联网是围绕着少数具体范例设计的，并且仅考虑了为数不多的几个应用程序，但它仍然具有相当强的可扩展性。通过稍后将用到的几个实现服务质量（Quality of Service，QoS，一种支持特定容量的“数据管道”的能力）的标准化方法的实例来展示这种可扩展性。

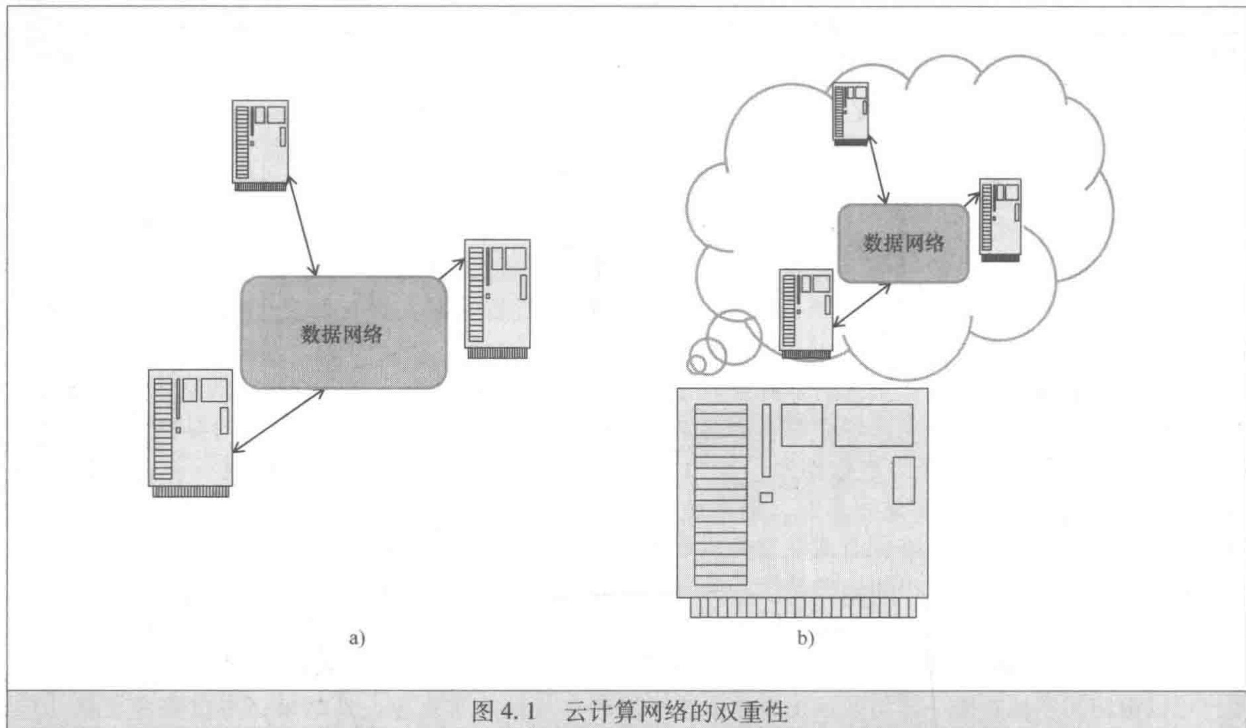


图 4.1　云计算网络的双重性

一个重要的问题是 IP 网络中的寻址和路由问题（应当作为单独的一节内容，虽然在这方面有专门的参考书目）。之后，将用一节来专门介绍多协议标签交换（Multi - Protocol Label Switching，MPLS），该技术通过有效地引入了一种基于分组交换的综合电路交换技术，几乎跳出了 IP 的范畴。

基于前面讨论的功能，为下一步虚拟化引入了一个新的维度。该维度涉及网络虚拟化，而不是计算机虚拟化。企业可以使用一组机制从运营商的网络或互联网中打造一种网络来取代建设单独的专用网络，所打造的这种网络看起来像一种专用的私有网络，即虚拟专用网（Virtual Private Network，VPN）。公共电话网已经提供了这种功能（参见本章参考文献［3］了解相关细节和历史），为企业提供了一种独特的编号方案，以便电话呼叫始终能够通过相同的方式来完成，无论呼叫者在相邻的办公室，还是分别处于不同的国家中。

在数据网络的情况下，在这一点上，准确地说，电话已经成为数据网络的另一种应用，因此，我们正在步入一种情况，除了数据网络之外，真的没有任何其他的网络。图 4.2 描述了几种 VPN 变体。图 4.2a 所示的专用网，由企业完全拥有。通常，这样的网络覆盖了单个企业园区。如果一个企业有两个园区，这两个网络可以通过一种“管道”实现互连，如图 4.2b 所示（顺便提一下，这是前面讨论过的 NIST 参考架构中描述的云端运营商提供的一种互连方式）。图 4.2c 描述了一种场景，企业的不同部分（园区网络甚至个人用户）通过运营商公共网络或互联网来构建 VPN。

有关该问题的进一步发展，本书另起一节，介绍了软件定义网络（Software - Defined Network，SDN），该技术为网络运营商在数据分组路由和转发的集中决策方面，提供了更多的控制权。

和上一章一样，本章最后也以安全性的问题收尾，概述了网络安全方面的内容，只不过这次讨论的范围仅限于网络层的安全性。



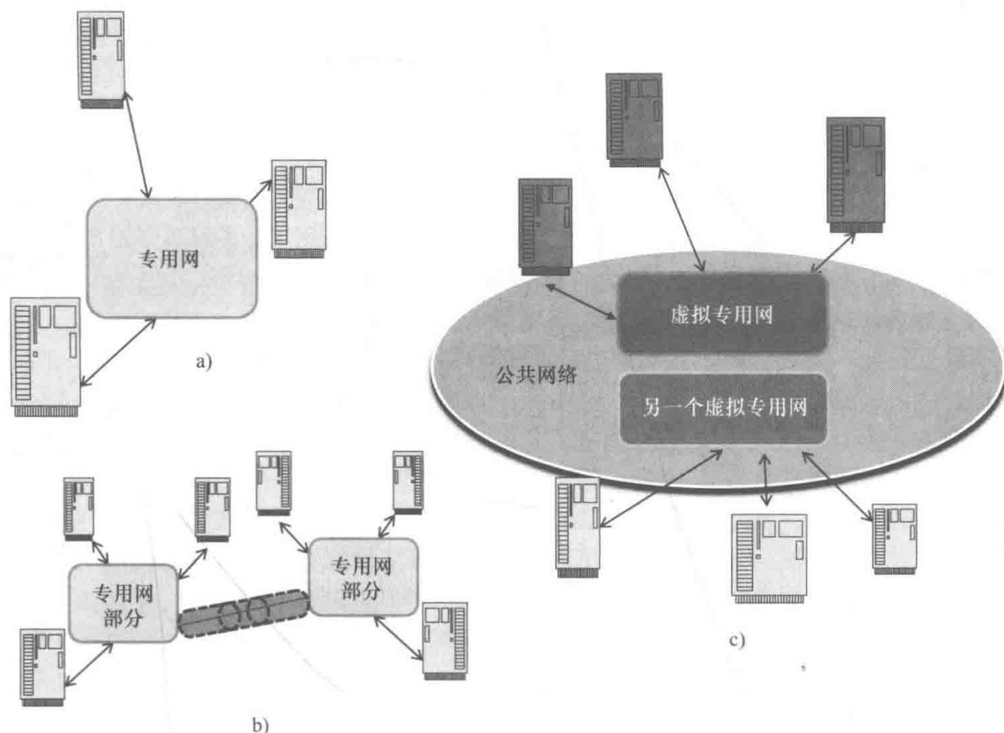


图 4.2 专用网和虚拟专用网

4.1 OSI 参考模型

OSI 参考模型（首次发布于 1984 年，随后在 10 年后进行修订[4]）成功使用了一种分而治之的方法解决了在不同主机上运行的两个进程之间彼此通信的复杂问题。尽管许多实际的 OSI 标准最终被放弃，以利于互联网标准的发展（参见本章参考文献［5］，该文献很好地解释了其中的原因和这项工作的历史），但该模型仍然在用㊀。图 4.3 强调了该模型中的关键方面。首先，端点被构造为七层实体，每层都是一个独立的模块，负责特定的通信功能。

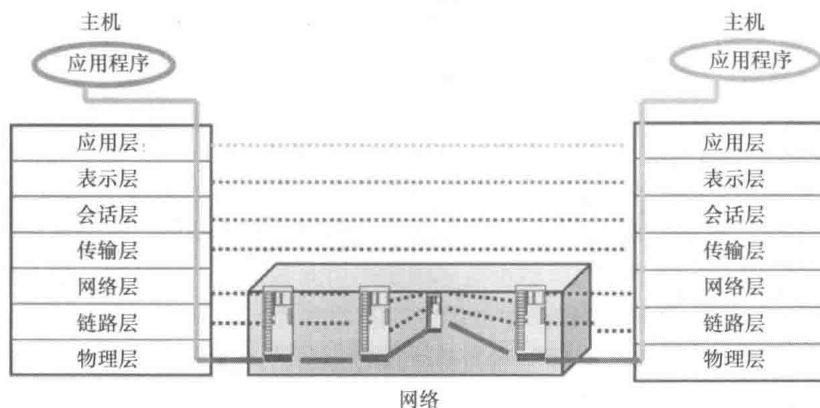


图 4.3 OSI 参考模型

㊀ 为此，很多 OSI 概念甚至协议最终也被用在网际协议集中。

该模型包含两个方面：一个是机器间通信；另一个是单个机器内的处理。在本节的剩余部分内容中，将论及这些方面，然后对每层分别给出相应的功能介绍。

4.1.1 主机到主机通信 ★★★

每层作为单独的模块，被设计成直接与其对应的层进行“通话”（如图4.3中的虚线所示）。一组与具体层相关的消息及定义其顺序的规则被称为协议。例如，文件传输（或许是最早的数据通信应用）可能需要一组协议，用来包含复制给定文件的消息以及一组操作状态消息（包括报错消息）。

虽然消息的实际传输由底层完成，终止于依赖物理媒介传输数据的物理层，但从外部来看，“直接的”层内消息（称为协议数据单元）被完好无损地提供给端点的接收模块。

端点实体通过寻址方案进行标识。因此，应用层的端点通过应用程序定义寻址方案进行标识，会话层的端点通过会话层寻址方案进行标识，以此类推。保持模块化的主要规则是，给定层不应当知道其以下层次中发生了什么，只需了解层间接口定义的内容（后面，在讨论网际协议时，将会看到，同其他规则一样，这项规则将被打破）。

不同层次之间的端点含义是不同的。从应用层到会话层，对话者是通过应用程序实体在模型中表示的应用程序进程。对于传输层，端点是机器本身。但是网络层完全不同，如图4.3所示，它通过多台机器作为中继传输协议数据单元，直到该数据单元到达预期的接收者主机。

这种交付模式值得特别注意，因为它有两种竞争模式。一种模式，称为无连接。在这种模式中，网络像邮政系统一样工作。一旦具有接收者地址的信封被投入系统中，它就可能会通过多个中继站，直到到达服务接收者的邮局。人们以往（也许仍然在）通过信件进行象棋游戏。在游戏的每一个回合中，通信者在信件中写下其将要走的下一步棋，然后发送给对方，并等待对方的动作。需要注意的是，信封传递的路线可能会因信封的不同而有所不同，但都需要遵守邮件接收规定，以及受制于可用的运输途径。通过信件下棋是一件缓慢的事情。

另一种模式，称为面向连接，类似于电话。这种模式典型的应用示例是电话会话。在20世纪的传统公用电话交换网（Public Switched Telecommunications Network，PSTN）中，电话交换机在电话机之间建立了一个端到端的路由（称之为电路），连接两部电话。一旦建立起来，该电路将持续保持，直到会话结束。此外，PSTN可以根据客户的要求建立半永久的电路，连接客户企业的各个部分，从而建立一种VPN。这种模型是天然的物理“在线”连接，但它也被应用到了数据通信中，即在网络层定义了虚拟电路㊀。与无连接模式相比，面向连接模式的优点是业务通信相对简单（即保证了端到端延迟的上限及其方差，称为抖动）。缺点是这种模式需要资源来维持电路，从而导致潜在的资源浪费。回到前面国际象棋的例子，如果玩家使用电话代替邮件，该游戏的进行速度必然加快，但是其所需的电话费用必然比邮件的费用高得多，因为在玩家思考其下一步棋如何走时，也需要保持电话线路的连通。

将多次回到这一话题。正如将看到的，这两种模式在发展结合的过程中，仍然存在。本章参考文献［3］对这些模式在电话网络和互联网络中的发展进行了详细的介绍和论述。最后需要注意到，在OSI参考模型下，图中看到的包含网络操作的中间这些机器的最高层是网络层。将这些机器称为网元（网络元素）。通常，网元也被称为交换机或路由器——其中的区别来自于相应的模式：面向连接或无连接。

最后，链路层和物理层都可以处理单个物理链路，从而连接两个网元，或网元和端点主机。

4.1.2 层间通信 ★★★

在教授数据通信和操作系统方面的经验证明，对计算模型的理解（即应用程序进程上下文中什么被调用，以及在哪些环境下被调用）是全面理解数据通信及其中涉及的每个协议的关键。

应当注意到，模型并不意味着实现，尽管很多实现都是基于一个或另一个已有的模型产生的。最终，对抽象思想具体如何实现的思考这一思想的关键一步。对于这些标准规范，将朝着将其付诸实现

㊀ 再次建议读者阅读本章参考文献［5］来获得有关这些技术和协议的详细信息。



的方向迈出这样的一步。

根据OSI模型，每一个层次都为其上一层定义好了服务。该服务是通过在层之间传递信息实现的一组功能。

这里，关键的问题是通信进程如何与图4.3中的层进行交互，以及这些层如何在单个机器（主机）上彼此交互。

OSI规范指出，为了将协议数据单元发送到对端，所在层需要向其下层发送请求。这些层严格按照分层结构进行交互，而不会绕过它们的排列顺序。（从对端层）到来的协议数据被携带到下层发出的指示中。因此，每一层从其上层接收请求，并将其自己的请求发送给它的下一层；类似地，每一层从其下层接收指示，并将其自己的指示发送给它的上一层。虽然这种模式比较简洁，但它是比较抽象的，因此可能会带来完全不同的实现结果。

先从一个简单但相对低效的实现开始，其中每一层作为一个单独的进程实现[1]，并且使用进程间的通信传递消息。这会带来一种相当慢的操作，并且存在内在的困难。由于指示和请求是异步的[2]，因此简单的实现（可能取决于给定操作系统的消息传递能力）至少需要每一层有两个进程（或单独的进程中有两个线程）。一个进程（或线程）执行等待指示并将其向上传播的紧凑循环，另一个进程等待请求并将其向下传播。稍微好点的方法是等待（或者从上一层，或者从下一层）消息，然后相应地处理它。

在图4.4的帮助下，描述一个有效的计算模型的一种方法是，将层定义为带有两种方法的类：请求和指示，以及由层的功能需求决定的适当的数据结构和常量。请求方法仅由上一层调用；指示方法仅由下一层调用。

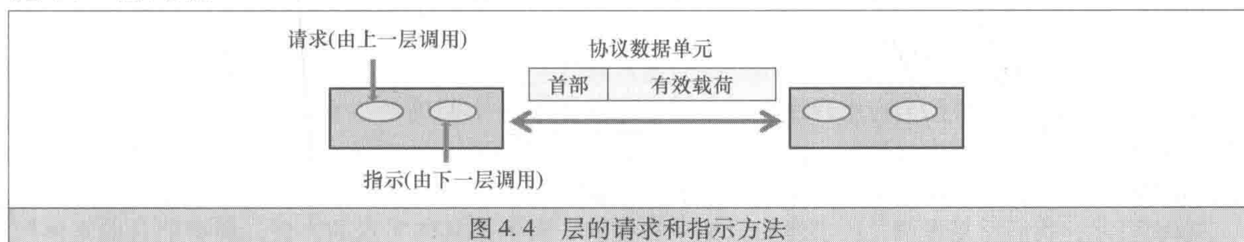


图4.4 层的请求和指示方法

根据OSI模型，在主机和网元上，每一层均需要有相应类的对象被实例化。对象按照该层协议规范的定义交换协议数据单元。当某一层发送协议数据单元时，会在上面附加一个首部，来进一步描述首部后面的有效载荷。然后，调用其下一层的请求方法，将该协议数据单元传递给它。在计算模型中，请求动作开始于应用程序进程本身，因此，所有后续的调用都是在该进程的上下文（应用程序实体）中执行的。

从上一层对下层请求方法的调用，将在某一时刻导致下一层对相应层请求方法的调用。不过，这并不一定都会产生这样的动作。例如，在调用下一层的请求方法之前，该层的请求行为可能需要将数据累加到某一点。相反，当调用请求方法时，可能会在下一层导致多个请求被调用，因为有可能需要将一个大的协议数据单元在下一层分解成多个较小的协议数据单元（例如，一个巨大的文件可能无法使用一个协议数据单元发送，因此将这样的文件从一个机器复制到另一个机器的请求，往往会导致多个数据单元被发送。稍后将看到更多的例子）。

同样的考虑也适用于指示方法的调用。正如在较高层发送的消息，可能需要在下一层被分解成较小的数据单元进行处理，因此接收到的消息在传递给上层之前可能需要进行重新组装。

我们知道，在这种请求链中的第一个请求是由应用程序进程调用的。但是第一个指示是由谁调用的呢？当然，就是那个用来监听设备从网络接收数据的部分。正如我们所知，CPU可以监听设备信号，这表明第一个指示方法应该由相关的终端服务例程调用，其余的调用都从那里传播。

---

[1] 已经看到了这种实现中的情况，产生的性能非常糟糕，以至于该项目最终被废止。

[2] 当进程需要发送数据时，会产生从应用程序进程到应用层的请求，并且很容易看出，其余的请求如何链接到原始的请求。然而，这些指示通常是由从网络中接收的消息引起的。

现在进入设备主题。应当注意的是，链路层和物理层通常都是以硬件的方式实现的，网元中的部分网络层也是如此。因此，实际面向设备的接口可能需要保持在上层。而且，从互联网的讨论中可以看出，一些应用程序完全跳过了上面几层，所以这种应用程序进程可能需要面对应用层以下的层，从图 4.5 对整个模型的描述中可以体现这一点。

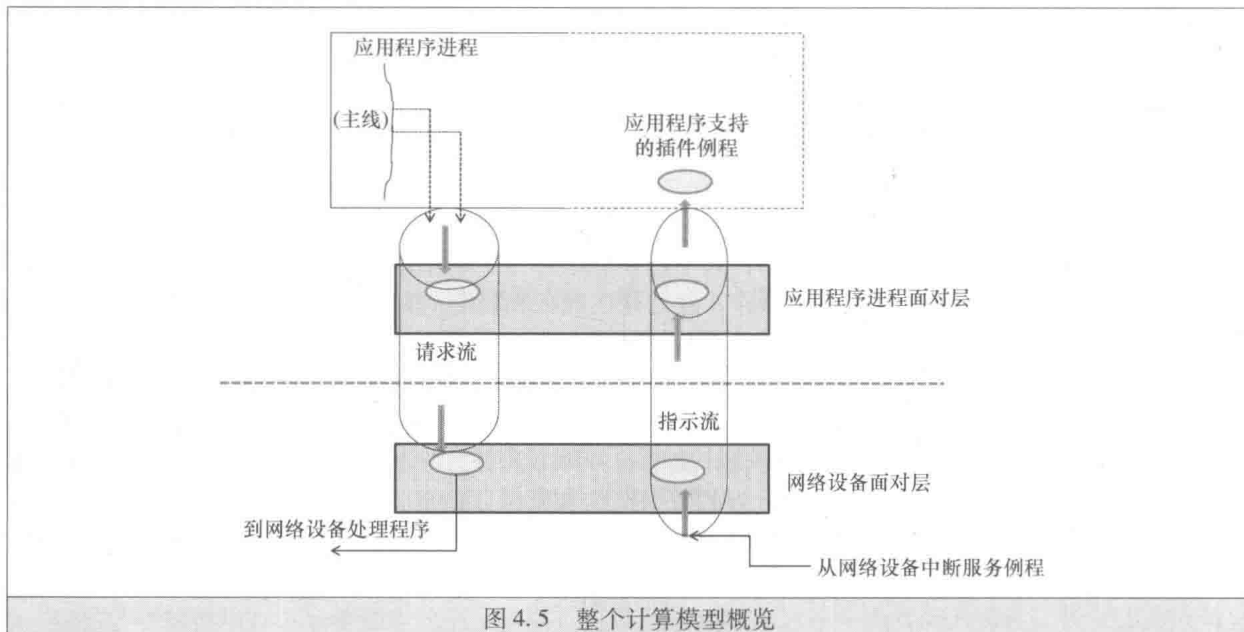


图 4.5　整个计算模型概览

需要注意的是，该模型为应用程序提供的指示方法给了一个明确的位置（省略了“进程”一词，因为进程被定义为应用程序代码的主线）。指示方法由应用程序作为插件导出，因此最终被调用不是通过应用程序进程，而是由操作系统作为中断处理的一部分。

总而言之，我们反复强调，所描述的只是一种模型，而不是具体实现的方法。模型的目的是阐明概念的所有方面。在 OSI 层间交换的情况下，主要问题是指示的异步性质。事实上，很多年前，本书的一位作者曾是 Sperry Univac 公司团队的成员，该公司就是使用前面介绍的这种方式，在 Varian 数据机（Varian Data Machines）小型计算机上实现 OSI 协议族变体。准确来说，链路层和物理层是在硬件（单独的板子）上实现的；其余的部分是通过对象库来实现的㊀。不过，这种实现是由操作系统中没有进程间通信机制决定的。

4.1.3　层的功能描述 ★★★

接下来将从应用层开始，按照自顶而下的顺序进行描述。毫不奇怪，后者的标准描述没有太多的内容。不过，除了数据传输之外，其主要功能包括通过适当的寻址方案识别对话者的身份、参与通信的认证、授权以及对 QoS 参数的协商（将在 IP 网络的背景下详细讨论其中的最后一点）。

还有一个重要功能是建立抽象语法上下文，这涉及数据结构（最初定义在高级语言中）到比特位字符串的转换。这样做的必要性源自于早期的数据通信，从字符编码开始。IBM 终端使用扩展的二进制编码十进制交换码（Extended Binary Coded Decimal Interchange Code，EBCDIC），这是 IBM 操作系统的通用语言，而 Bell 系统电传则使用美国信息交换标准码（American Standard Code for Information Interchange，ASCII），其起源于电报。小型计算机操作系统采用 ASCII，它也是虚拟电传终端使用的标准。

字符表示仍然是一个重要的问题，它促使了与互联网国际化有关的新标准的出现，但是这个问题只是数据结构表现形式的冰山一角。首先，这些机器的架构在八位比特组（一个字节）的比特读取方

㊀ 准确地说，当时没有面向对象的语言使用，所以严格来说，实现的是一种子程序调用库，这些库是单独的模块编译的，每层一个模块。每个模块对应着一个程序，具有自己的数据结构和程序库。数据仅对模块内的程序可见。



式中有所不同，普遍有两种选项——从左到右和从右到左。而且，计算机字（word，无论16位还是32位的）中的字节顺序在不同的架构之间也是不同的，有符号整数和浮点数的表示也是一样。因此，需要一种对网络连接比特字符串进行构建（由发送者）和解释（由接收者）的机制，这正是抽象语法上下文的用处所在。

最后，应用层的OSI标准在无连接和面向连接的操作模式之间是不同的。安全功能，以及负责建立错误恢复和同步的功能，仅被分配给面向连接的模式。

从表示层开始，所有层都为其直接上层提供了特定的服务。表示层的服务是识别所有可用的传输语法（即网络连接上比特位字符串的布局），以及对所使用的传输语法的选择。除此之外，表示层为会话层提供传递方式和途径，通过将传输语法映射为抽象语法来有效地对会话有效载荷进行翻译转换。协商传输语法是表示层的功能之一。这里有必要看一下计算方面的知识：传递到表示层的每一个携带有效载荷的请求，都会导致对会话层请求的调用。相反，传递到会话层的每一个携带有效载荷的指示，也都会导致对表示层指示的调用。两者被同步处理，表示层负责翻译转换有效载荷[㊀]。

会话层在两个进程（通过它们各自的表示实体）之间通过会话地址提供双工连接。会话层服务包括连接和数据传输的建立和释放，以及连接重新同步，管理和相关的异常报告。此外，还有一个令牌管理服务，可以让通信进程轮流执行控制功能。会话层可以为较短的会话协议数据单元提供一种特殊、加急服务，用以支持服务质量需求[㊁]。

上述服务只与面向连接的模式有关。在无连接的模式中，会话层服务只不过是到传输层的中间传递过程。

传输层实际实现了这些功能，用来支持为上述各层定义的服务。OSI设计的宣称目标是优化“对可用网络服务的使用，从而以最低的成本来为每个会话实体提供所需的性能”。

传输层为会话层提供的服务，实际上与其为会话层定义的那些服务不同，但是传输层是唯一正在从事这项工作的层。一个有趣的功能是会话多路传输，如图4.6所示。

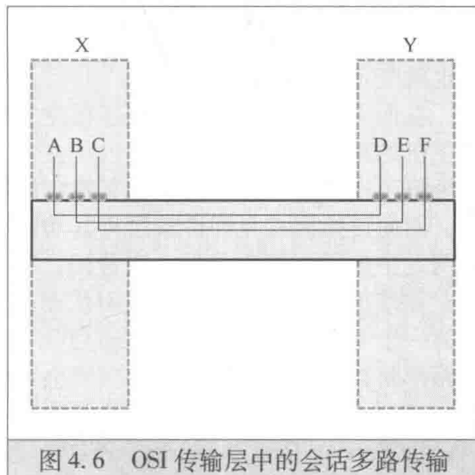


图4.6 OSI传输层中的会话多路传输

这里有三个会话（AC、BD和EF）被多路传输到主机X和Y之间传输层连接之中（不幸的是，从OSI标准的一开始，该模型就一直没有被实现，而是始终保持互操作性的方式。正如所看到的，互联网模式完全没有考虑会话层和传输层之间的差异）。反过来，传输层可以将传输连接多路传输到网络层连接上。其中，后者在必要的时候也可以被分开。

传输层的其他独特功能还包括传输协议数据单元序列控制，传输控制数据单元的分片和级联，流量控制，以及差错检测与恢复，所有这些功能都是通过加急数据传输和服务质量监控来增加和加强的。在面向连接的模式中，这组功能保证了一种可控、端到端无差错通信管道的存在[㊂]。

实现这一点可以说是一项重大的壮举，因为无连接的网络可能会导致传输的数据单元无序或完全丢失。传输层连接的建立和拆除本身就是一个意义重大的问题（从第6章引用的参考文献［5］中可以看出）。

一旦连接建立，传输层必须历数它所发送的所有数据单元[㊃]，然后跟踪来自对端的包含有接收数据

㊀ 我们将看到，美国国防部高级研究计划局网络（Advanced Research Projects Agency Network，ARPANET）协议的设计人员（后来制定了网际协议集）放弃了表示层。但是，该功能被引入到以后的互联网应用层协议中（当越来越多的应用程序需要它时），为抽象传输语法提供了多个不同的选项。

㊁ 最初，互联网不接受服务质量这个概念，但稍后将会看到，最终互联网还是接受了这个概念，甚至使用了OSI中的术语。不过，互联网将对服务质量的支持放在了传输层和网络层中。

㊂ 在面向连接的模式中，上述功能中只存在端到端差错检测和服务质量监控这两个功能。

㊃ 自然，这些序号有一个定义的大小，因此要将它们对某个数M进行取模数运算，但是这样又带来了另一个设计问题：必须要确保在任何时间没有具有相同序号的消息是未完成处理的。

单元序号的确认。所有未得到确认的协议数据单元均需要被重新传送。在接收侧，对于带有非连续序号的每两个协议数据单元，传输层必须等待直到收集到所有数据单元能够圆满无漏地填充这两个非连续序号之间的间隙时，才能将该数据传递给会话层。在调校多个参数的过程中，用到了复杂的试探法，尤其是定时器取值，并且由于在数据传输过程中，对一个或两个主机都崩溃的情况需要使用恢复程序，这就使整个方案进一步复杂化。

差错检测与纠正需要计算传输协议数据单元的标准哈希（hash）函数（例如，校验和）。发送方发送计算后的量值（通常将其作为首部的一部分），接收方在接收的消息上计算这一函数，并将其计算结果与首部中包含的发送方计算的量值进行对比。在最简单的情况下，没有差错纠正被调用。如果比较失败，接收的消息将被丢弃，并且接收方不会为该消息回送确认。没有确认最终会导致发送方对这一消息进行重新发送。

差错纠正使用特殊的哈希函数，它的结果会携带“冗余”比特，以便使有效载荷中损坏的比特位被重新构造成为可能。循环冗余校验（Cyclical Redundancy Check，CRC）是一种典型的用于差错校正的哈希函数，CRC 在硬件中实现的性能最佳。我们注意到，在互联网中，传输控制协议（Transmission Control Protocol，TCP，一种简单的校验和例子）用于此目的，通过重新传输实现差错控制。

网络层为传输层提供的基本服务是传输实体之间的数据传输。因此，目的是屏蔽传输层通过网元中继其协议数据单元的细节。网络层协议数据单元被称为分组。

在网络中，从主机到达的分组可能会被分解成很多较小的块，通过不同的路由进行发送（想象一下，需要通过邮局将 1t 重的邮包寄往海外。很有可能，会被要求将邮包拆分成多个较小的邮包。之后，一些邮包可能会以船只的方式送达目的地，而其他的会通过飞机送往目的地）。这几乎就是无连接网络层模式工作的原理。

在 OSI 模型中，网络层支持面向连接和无连接两种操作模式，但这两种模式是完全不同的，这使得 OSI 网络层相当复杂（在互联网模式中，面向连接的模式最初被人们完全拒绝，但一旦引入，网络层也就变得复杂了）。

在面向连接模式中，网络连接建立阶段会创建穿越网络的端到端虚电路。换句话说，在网络中一个主机的路由在连接持续期间内建立一次，所以给定虚电路上所有分组都将穿过相同的网元。这有助于为每个连接设计网络并保证其特定的服务质量参数，例如吞吐量和端到端传输延迟（及其变化）。此外，面向连接模式允许传输层将其功能的大部分外包给网络层。这种模式的主要缺点是在连接期间绑定网元中的资源需要支付一定费用。在极为罕见的情况下，如果端点只需要交换一个短消息，则无须建立和维护大部分时间处于空闲状态的连接。然而，如果需要经常建立短连接，那么连接建立事件将成为一个不得不考虑的因素。还有一个缺点是单点故障：如果某个网元崩溃，则通过该网元的所有虚电路都将被断开，并且所有相应的会话将被终止。这些缺点对典型业务是没有影响的，正如基于一组 ITU－T X 系列建议（其中最著名的是 X. 25[6] 协议，其中对网络层进行了规范）的公共服务数据网（Public Service Data Networks，PSDN）所示。

这里的发展出现一点历史的迂回还是值得的。对面向连接服务的研究既不是学术性的，也不是某种标准化的工作。到 20 世纪 60 年代末，企业迫切需要将其在地理位置上相互分离企业部分的计算机连接起来。在当时，已经拥有了庞大网络（企业实际上被连接到这种网络上）的电信公司，是唯一能够满足这种企业需求的候选机构。

在 20 世纪 70 年代，紧跟着公共交换电话网（Public Switched Telephone Networks，PSTN）服务之后，人们就开发了类似的公共数据网（Public Data Network，PDN）服务。Datran 数据拨号服务于 1974 年在美国运行，之后由南太平洋通信公司㊀提供。Datran 数据拨号服务拥有一种虚电路，它的误码率不逊于传输 10^7 个比特位错误一位[7]。之后，PDN 服务持续增长。

在 20 世纪 80 年代，电话公司开始开发综合业务数字网（Integrated Services Digital Network，ISDN），该网络可以提供将电路交换语音与基于 X. 25 分组交换数据组合的管道通信方式，其中后者将通过 PDN

㊀ 作为 Sprint Nextel 公司的前身，南太平洋通信公司（Southern Pacific Communications Company）是南太平洋铁路公司（Southern Pacific Railroad，SPC）的一个单位，它拥有大量的通行权。有句话说，运输业与电信业始终相互竞争，但是在 SPC，这两个行业是统一的。



虚电路进行传送。

在20世纪90年代，ISDN被部署在几个欧洲国家和日本，并且其标准化工作进入宽带ISDN阶段，其中网络层采用X.25类的帧中继和异步传输模式（Asynchronous Transfer Mode，ATM）协议，尽管基于不同的技术，但两者都提供了虚电路。这些发展在20世纪90年代中期达到顶峰，但在20世纪90年代后期，万维网以及基于此的互联网显然赢得了最终的胜利。当时，互联网标准中的网络层已经没有地方采用向ISO这样的面向连接的模式，尽管如后所述，人们开始制定类似的东西。

在无连接的模式中，网络层将其处理的数据传送到目的地，通常只能做到尽力而为。一旦网元接收到分组，它只会将其传递给它的相邻节点。选择相邻节点的决定由分组的目的地址和网络层的路由表决定，该表可以使用这样或那样的路由算法来动态地进行配置或构建。路由算法计算到达目的地的最短路径。这里的“短”不是指地理距离，而是指由分配给每个链路的权重定义的距离（这里的指标范围从链路容量到其当前的有效载荷）。路由需要真正的分布式计算：所有的路由器通过广告它们的存在并传播关于它们链路的信息来参与该计算（再次，建议读者参阅本章参考文献［5］来更加全面地了解有关这一主题的详细内容，包括参考书目）。在最简单的情况下，采用最初在ARPANET中考虑的并被称为洪水的技术。使用这种技术，不需要路由表。网元简单地将它所接收的分组重新发送到它的每一个相邻节点上，当然除了作为该分组来源的相邻节点。

无论是无连接还是面向连接的，到目前为止在这些所考虑的层中，网络层都是第一个基于中继而不是点对点通信进行操作的层。事实上，在早期的网络（和OSI模型的早期版本）中，网络层是唯一一个明确基于中继的层。因此，OSI明确禁止在应用层和网络层之间的层使用中继。

现在已经为讨论OSI参考模型下面几层做好了准备，将来会在链路层和物理层之间进行介绍。这就是为什么OSI参考模型后期版本的标准指出：“物理层的服务是由底层媒介的特性决定的……”，但是链路层仍然可以囊括所有这些服务。这本身并没有什么问题，因为链路层具有明确和独特的功能，但是出于教学的原因，从底层媒介特性的角度来阐明这个功能的各个方面要好得多。

回到数据网络的早期阶段，物理连接（例如，双绞线，或从电话公司租赁的电路，或关注的微波链路，或卫星链路）都是点对点的。因此，沿着线路的数据传输可能有着明显的误码率，特别是当涉及长距离模拟线路时。模拟传输需要在端点有调制解调器，并且它的误码率要比数字传输高出两个数量级。

同样，先来回顾一下历史。在20世纪80年代和90年代初期，只有一个用于个人电话公司入户家庭用户的模拟传输业务。然而，数字业务可用于企业用户㊀。西联宽带交换业务（Western Union Broadband Exchange Service）于1964年在得克萨斯州成功推出。根据本章参考文献［7］，在20世纪70年代初，Multicom和Caducée在加拿大和法国分别推出了类似的业务，提供“速度高达4800bit/s的全双工点到点交换业务，误码率低至之前一般只有专用线路上才能实现的程度”。实际上，这些服务是物理层PDN服务的开始。

物理层服务本质上是一个原始比特流。先来看一下点对点连接。在这种情况下，链路层功能与传输层功能几乎相同：需要在两个端点之间建立连接，然后在站之间交换协议数据单元。在发送端，链路层将网络有效载荷封装成帧，帧被逐个比特地送到物理层。帧中有效载荷伴随着纠错码。正如观察到的，连接可能出现故障，尤其在两端采用调制解调器的长距离模拟拨号线路上。因此，纠错是这种媒介的链路层提供的基本功能。

一些人对链路层上的纠错功能进行抨击，因为他们认为这项工作由传输层提供就足够了。图4.7阐述了对这种观点的反驳。这里，当采用纠错时，在故障链路上通过本地重传可以修复的内容，如果在没有采用纠错时，只能通过在整个网络上进行重传修复。当消息路径中存在多个故障链路时，情况会变得更糟。

与此同时，到20世纪70年代中期，物理媒介发生了重大变化。

从LAN的发明开始[8]，广播媒介大量出现。图4.8描述了4种类型：总线、星形、无线LAN实际共享单个媒介，以及环形LAN组合了一系列位转发器。

㊀　实际上，这是ISDN在美国部署的一个严重障碍，ISDN需要数字用户线。

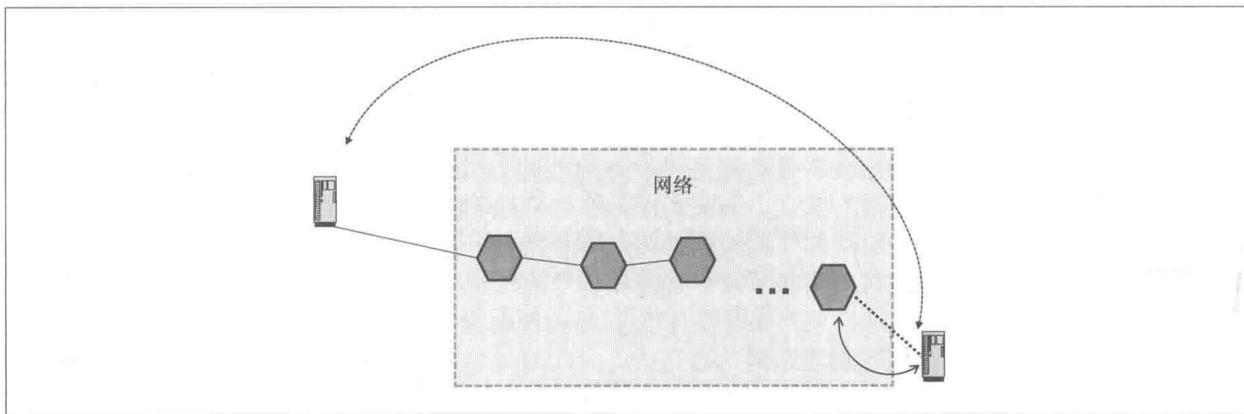


图 4.7　链路层纠错的情况

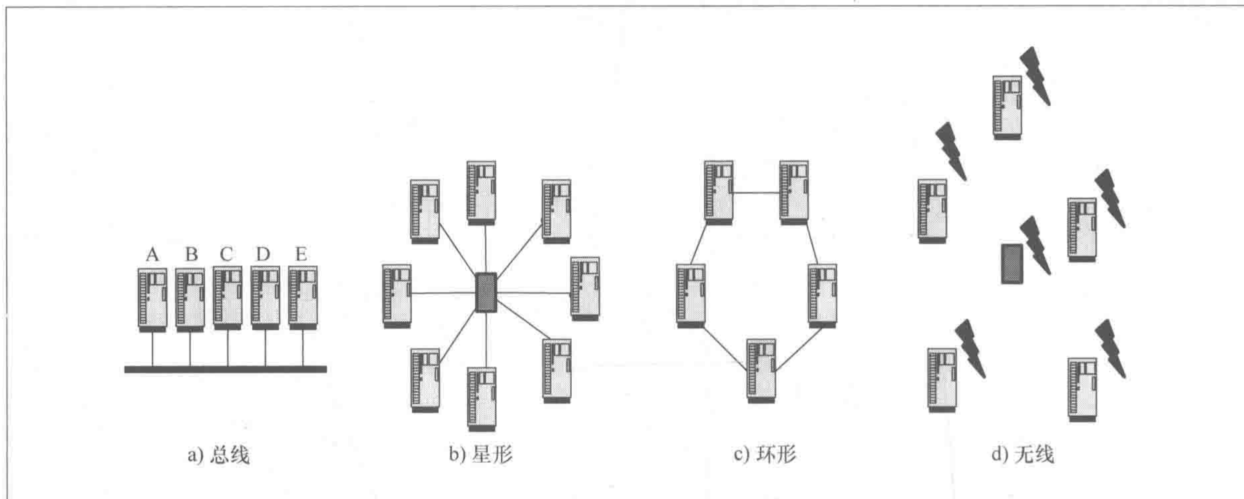


图 4.8　广播媒介配置

这种发展以多种方式改变了链路层的需求。首先，广播需要唯一的链路层标识，必须在每个帧中明确携带。其次，LAN 以高得多的比特速率运行，从而实现语音和视频业务，因此，设计服务质量参数来根据具体业务优先级分配带宽是有意义的。第三，广播的本质就是把一个基本的网络层功能［定位消息的接收者（当然是在 LAN 边界之内）］推向下层的链路层和物理层。第四，在接受一些网络层功能的同时，链路层可以轻松地将维护连接的功能委托给传输层。为此，链路层中的“链路”一词几乎成为一个使用不恰当、不准确的词。第五，共享媒介需要一组特殊的访问控制功能，从而形成并产生了不同的链路层子层。第六，广播媒介（而不是点对点链路）需要实施安全机制来保护数据。

在第 6 章讨论存储方面的内容时，将再次回到局域网的主题上来。

4.2　网际协议族

Vint Cerf 和 Bob Kahn 在 1974 年的一篇论文[9] 为网际协议族奠定了基础㊀。论文本身（比 ISO 的 OSI 标准项目的创立早了三年）就是一个清晰简洁的技术方法。文中列举了当前存在的各个方面的网络问题，推断了总体解决方案的要求，并通过系统应用简约方法提出了一种解决方案。

本章参考文献［9］中提出的最初动机和产生的要求是值得人们遵循的。这篇论文认为，在单个分组交换网络中运行的数据通信协议已经存在，然后列举并审视了 5 个网络互操作方面的问题。

第一个问题是寻址。认识到存在具有“接收机不同寻址方式”多个网络，论文强调需要一种可以被每一个单独网络理解的“统一寻址方案……”。

㊀ 顾名思义，该论文介绍了所有分组交换网络（现在的和未来的）互联的解决方案。

第二个问题是最大分组尺寸。因为给定网络可以接受的单个数据单元的最大尺寸，在不同的网络之间是不同的，并且最小的这种“最大尺寸”可能是“不切实际的”，以至于不能在标准上达成一致，文中给出了“允许跨越网络边界将数据重新格式化成更小部分的需求程序”的可选方案。

第三个问题是针对确认帧的最大可接受端到端延时，该延时超出则可以认为相应的分组丢失。这种延时取值在不同网络之间是不同的。因此需要“精心设计并开发网络定时程序，以确保数据可以通过各种网络，成功地被交付”。

第四个问题是数据突发和丢失，这需要“端到端的恢复程序”来进行恢复。

第五个问题是网络在其“状态信息、路由、故障检测和隔离”中的变化。为了处理这个问题，“必须要在通信网络之间启动各种协调工作”。

为了解决第一个问题，人们提出了一种统一的网络地址，其中“TCP 地址㊀”包含网络标识符和 TCP 标识符，TCP 标识符反过来可以标识网络内的主机，而端口是通信进程的直接管道。

为了解决第二个问题，人们提出了网关连接两个网络的概念。要知道它连接的两个网络中的每一个，网关在它们之间通过诸如分片和必要时的重新组装来进行协调。其余的问题可完全交由端到端操作进行处理。该论文详细规定了基于法国 Cyclade 系统已经使用的滑动窗口机制的重传和重复检测过程。

在结论中，这篇论文要求制定详细的协议规范，以便可以通过实验来确定操作参数（例如，重传超时）。

在接下来的 6 年中，人们遵循上述提议为 ARPANET 制定了相应的协议规范。在此过程中，协议被分为两个独立的部分：网际协议（Internetworking Protocol，IP），负责处理分组结构和网络层的过程；传输控制协议（Transmission Control Protocol，TCP），负责处理端到端传输层的问题。虽然其他的传输协议也被互联网所接受，但在提及整个网际协议族时，TCP/IP 这个词已经成为一种常态标准。

1981 年，稳定的标准 IP（IP 版本 4 或 IPv4）在本章参考文献 [10] 中对外发布，作为美国国防部的网际协议㊁。该协议在今天得到了广泛的应用，尽管最新的标准 IPv6 目前正在部署，而且，IETF 仍在努力开发新的标准版本。

4.2.1 IP——互联网的黏合剂★★★

图 4.9 显示了 IPv4 分组的结构，令人惊讶的是，尽管其中有些字段已经被人们重新解释，但是这一结构仍然保持不变，不久读者将会在后面内容中看到。

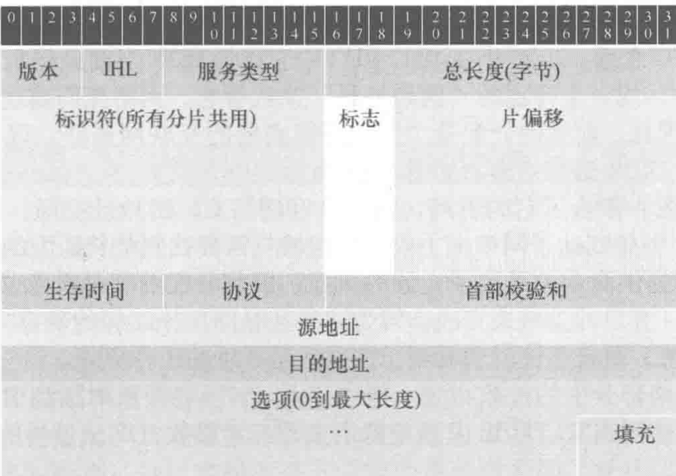


图 4.9 IPv4 的分组首部

㊀ 该论文中的缩写指的是“传输控制程序（Transmission Control Program，TCP）”，这个协议结合了今天所说的 TCP 和 IP。

㊁ 后来，互联网工程任务组（Internet Engineering Task Force，IETF）于 1986 年成立之后，将该文件作为 RPC（Request for Comments，征求修订意见书）791 对外发布。在本书的其余部分，将严格按编号引用 IETF RFC。

必须要理解这些字段，因为 IPv4 仍然是互联网协议的主要版本（并且在云中使用）。从互联网寻址问题的 IPv4 解决方案开始，因为这个问题首先列在 Cerf 和 Kahn 先前讨论过的最初愿景中。每个 IP 分组都有一个源地址和目的地址，它们具有相同的类型——一个 32 位字符串表示对：

<网络地址，主机地址>

需要着重强调的是，根据其定义，一个 IP 地址不是一个主机地址，因为对于一个主机来说，它不是唯一的。一个主机可以是多宿主的，驻留在多个网络上。在这种情况下，它所拥有的 IP 地址的数量与其所属的网络的数量一致㊀。类似地，作为连接网络角色的路由器，具有与其互连的网络一样多的不同的 IP 地址。

实际上，RFC 791 甚至允许仅与一个网络连接的主机也可以拥有多个 IP 地址："单个物理主机必须能够像使用多个不同的互联网地址，能够像多个不同的主机一样运行"。相反，使用任播（anycast）寻址（这在 RFC 791 时代甚至无法被人们所想像，但是之后将会遇到），为了达到负载均衡的目的，将共同的 IP 地址分配给提供相同服务的多个主机。

回到 IP 地址的逻辑结构上来。通过最初的设计，网络地址类（Class）标签分为 3 种：A、B 和 C 类。为此，RFC 791 规定：

"有 3 种格式或 3 类互联网地址：在 A 类中，最高位比特为 0，接下来的 7 位是网络地址，后面的 24 位是本地地址；在 B 类中，高位的前两位比特为 10，接下来的 14 位是网络地址，后面的 16 位是本地地址；在 C 类中，高位的前三位比特为 110，接下来的 21 位是网络地址，后面的 8 位是本地地址。"

首先，观察到类标签已经被放置在地址的开头。IP 字符串的解析将从地址类的确定开始。如果 IP 地址最左边的比特位是 0，则该地址类为 A；如果开始的前两位为 10，则为 B 类；如果前三位是 110，则为 C 类。这种编码方案的惯例是易于扩展的（例如，以后的 D 类——可以使用 1110 标签，被定义为组播地址㊁）。

其次，基于地址类的方案思想是拥有很多小型（C 类）网络，更少的 B 类网络（拥有更多的主机），以及只有 128 个巨大的 A 类网络。在当时，这种想法是合理的。图 4.10 描述了 1982 年由 Jon Postel 创建的互联网图谱㊂。

然而，大约 10 年后，IETF 发现了 3 个主要的问题，虽然都是意想不到的和前所未有的，但却是伴随着互联网的发展而产生的。第一，中等大小的 B 类网络的地址空间正在枯竭。第二，路由表变得太大，路由器难以维护，而且这种情况越来越糟糕。第三，整个 IP 地址空间正在耗尽。

最后一个问题从本质上来说，不是在 IPv4 的设计上能够被解决的，但是前两个问题可以通过去除类的概念来解决。因此，无类域间路由（Classless Inter - Domain Routing，CIDR）方案首次作为一个过渡解决方案（直到 IPv6 接管）出现，之后（因为 IPv6 尚未接管）作为一种不同程度的永久性解决方案。在连续发布 3 个 RFC 之后，2006 年在 RFC 4632 中它作为 IETF 当前最佳实践得以发布。

CIDR 摆脱了类标签，用一个特定的"网络掩码"来代替它，网络掩码描述了一种前缀，即地址的网络部分的确切位数。因此，前缀的分配是"旨在遵循底层的互联网拓扑，以便可以使用聚合来促进全局路由系统的扩展"。CIDR 聚合的概念如图 4.11 所示。

由其前缀标识的网络 A 聚合了它的子网（网络 B 和网络 C）的地址空间，网络 B 和网络 C 的前缀分别是 x0 和 x1。因此，要想到达子网中的主机，路由器只需要找到最长匹配的前缀即可。

到 2006 年，互联网提供商业务得到了充分的巩固，因此分配前缀是有意义的。所以，"前缀分配和聚合通常根据提供商 - 客户的关系来完成，因为这是互联网拓扑结构的确定方式。"当然，该策略能够后向兼容基于类的系统，而且很快就会看到，这也正是寻址的工作原理。

前缀由 IP 地址的网络部分中的位数指定。掩码以及"/"字符被附加到 IP 地址上。在这一点上，有必要将其在 IP 地址上体现出来。尽管 IP 地址的语义是一对整数，IP 地址传统上是按照逐个字节以点

㊀ 该网络（如今，它是一个 LAN）为主机提供了唯一的第二层地址，该地址被硬编码在 NIC 的硬件中。第 3 章介绍了与 NIST 安全要求相关的首字母缩略词，在描述设备模拟时就用了这一概念。为此，在 XEN 中对 NIC 进行了模拟，因为所有 VM 设备流量都是通过 IP "路由"到 dom 0 的。

㊁ 参见 RFC 5771。

㊂ Jon Postel 除了对互联网发展的许多服务外，还管理着互联网数字分配机构（Internet Assigned Numbers Authority，IANA），直到他于 1998 年逝世。

图 4.10　Jon Postel 在 1982 年的互联网图谱

分十进制的格式拼写的，如 171.16.23.42。掩码的拼写方式与其相同。考虑上述地址（属于 B 类），它的网络掩码是 255.255.0.0，因此以前缀符号表示的完整地址为 171.16.23.42/16。类似地，C 类地址 192.16.99.17 的掩码为 255.255.255.0，可将其指定为 192.16.99.17/24。

图 4.12 演示了如何将一个现有的 B 类地址在内部分解成 4 个子网。

在这一点上，我们必须认识到，通过快速发展的 20 年，我们已经偏离了 RFC 791 中规范的内容。现在回到图 4.9 中其余 IP 首部字段内容的讨论上。

第一个字段是版本，它定义了其余的首部结构。似乎这是 IP 首部实际资源中最浪费的字段。分配了 4 个比特位来指示协议的版本；这在实验中是有一定的帮助意义的，并且这些年来，只有两个协议版本——IPv4 和 IPv6。然而，IP 被设计成可以被人们永远使用，因此预计每 30 年左右，会有一对新版本的 IP 被人们广泛使用是有道理的。

首部长度（Internet Header Length，IHL）字段，顾名思义：网际协议首部长度，以 32 位字为单位。实际上，这是指向 IP 有效载荷开头的指针。

服务类型字段被设计为用来指定服务质量参数。有趣的是，尽管在这个（1981年的）规范中明确列出的应用是telnet和FTP，但分组优先级已经被人们所考虑，并且正如一些文档报告中所提到的，在某些网络中已经对分组优先级进行了实现。当时，主要的选择是在低延时、高可靠性和高吞吐量之间的三重权衡。然而，稍后这些比特位将被重新解释用于不同的服务；将在下一节中对这些新的解释进行讨论。

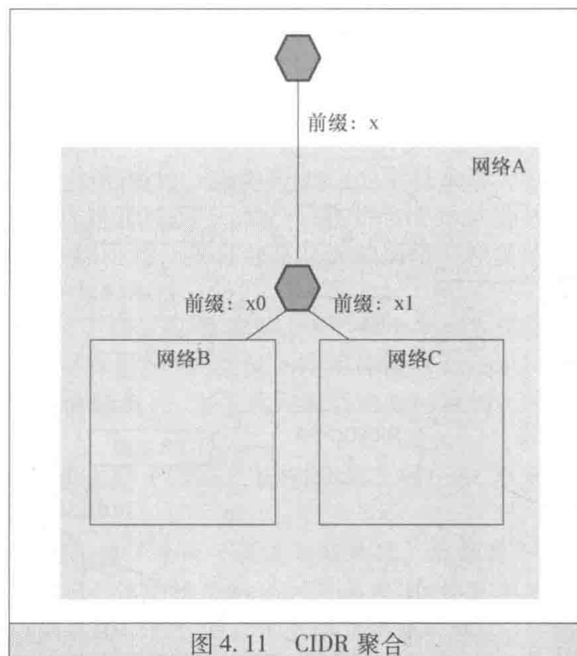


图4.11　CIDR聚合

总长度字段指定了分组的长度，以字节为单位（首部长度也被计算在内）。尽管该字段的长度为16位比特长度，但是只有在确保目的主机准备接受的情况下，才推荐主机发送长度超过576字节的分组。在撰写本书（英文原版）时，最大分组大小，即65535字节长度已被认为是受限的，但在1981年长分组的传输可以通过分片来实现。为此，接下来的三个字段用来处理分片事宜。

标识符字段所提供的值用于帮助数据报分片的组装（人们已经尝试对它进行重新定义）。标志字段（第一位仍被人们保留，未进行使用）包含1位的DF和MF字段，分别代表不分片和更多分片。其中，前者指示路由器丢弃分组（并报告错误），而不是对其进行分片；后者表示后面跟着更多的分片。片偏移字段是指向当前分片在原始有效载荷中位置的指针（因此，第一个分片的偏移值为0）。需要注意的是，IPv6完全取消了分片。取而代之的是，需要在接收分组的路径上发送分组。

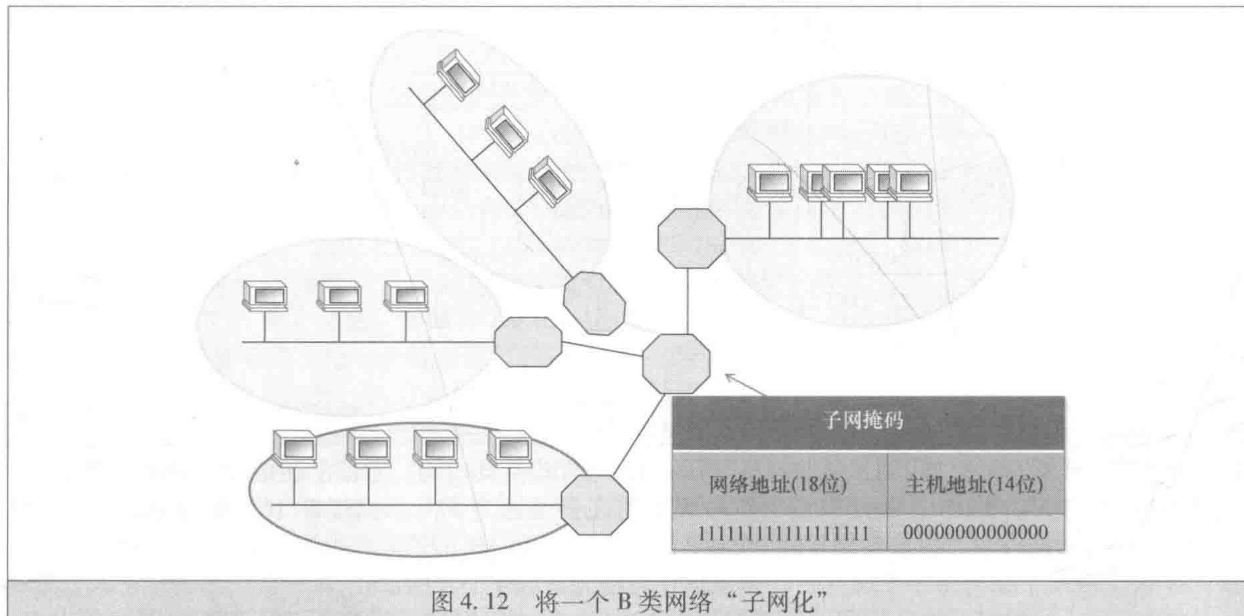


图4.12　将一个B类网络“子网化”

生存时间（Time－To－Live，TTL）字段，用于防止分组在网络中传输的无限循环问题。因为最初的互联网没有设想回路，互联网路由器在向分组目的地址转发分组时，采用“尽力而为”的做法。路由协议让路由器对网络中的变化做出反应并更改路由表，但是这种更改的传播可能很慢。因此，分组可能陷入循环。TTL的值用于决定一个分组是否处于“流浪”状态的。该字段是“可变的”，因为它的取值会被每一个路由器所更新。最初，该字段被设置为分组可在网络中传播时间的上限，并且每个路由器均应当修改该字段的值，将其值减去路由器处理该分组所需消耗的时间。一旦该值达到0时，该分组应当被丢弃。目前，这一字段的语义略有变化：以跳数来衡量，而不是以秒来衡量（因为以秒来衡量的方式难以处理和实现）。

协议字段指定了采用的传输层协议。回到在本章开头讨论的计算模型，有必要在主机接收分组时确定需要调用哪个过程/程序（或是哪个进程去发生这一分组）。

首部校验和在每一个路由器上计算，通过使用1的补码加法将首部的所有16比特字相加（大部分情况下该操作是在这些分组到达时进行的），并对相加和取1的补码进行计算。当分组到达时，如果没有错误发生，则在整个首部上计算的校验和必须等于0，因为首部已经包含了之前计算的校验和。如果结果值不为0，则丢弃该分组（需要注意的是，网络层只检查首部，而有效载荷的验证则是上层的工作）。如果路由器需要转发分组，则在改变TTL值之后，重新计算校验和。

在首部中，源地址和目的地址位于可选字段之前。其中，可选字段是可选的。最近还没有使用它（过去，可选字段的一个典型的用途是明确列出需要通过的路由）。最后，将分组填充足够位数的比特，使其长度始终保持为32比特的整数倍界限（以防可选字段没有以完整的32位字结尾）。

以上对IPv4首部结构进行了简要的概述。现在，应当清楚IPv4如何解决网络互连的原始问题了。在这个过程中，也提到了几个新的问题，其中包括IP地址空间的耗尽。我们没有讨论安全性问题，这是一个重大的问题，需要单独的一章（来提供并满足所有的先决条件）进行论述。还有一个大问题是对所谓的“服务质量”没有明确的支持。这将在4.3节中详细讨论；然而，现在提到它，因为下一步要解决的IPv6已经提供对IPv4的改进。

（基本的）IPv6首部结构如图4.13所示。

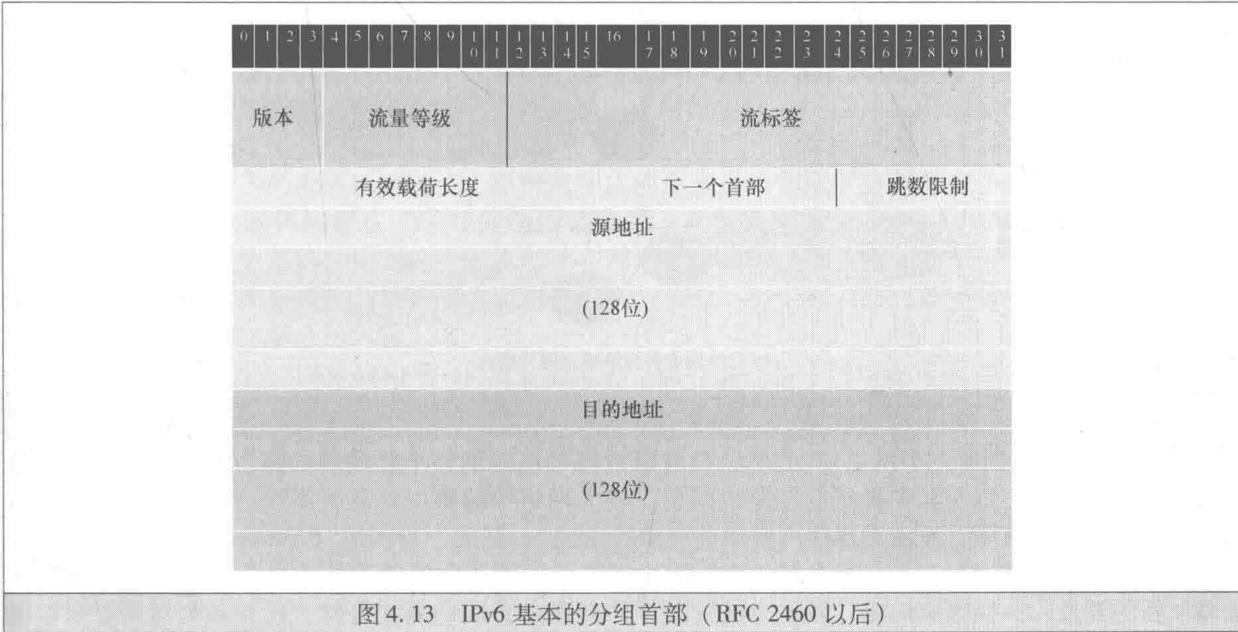


图4.13　IPv6基本的分组首部（RFC 2460以后）

粗略浏览IPv6的首部之后，或许最引人注目的是，它的首部似乎比IPv4还要简单。它确实是！一些字段已经被取消了，或将其作为可选字段，这使得它不仅更容易理解，而且能够更快地执行。IPv6的首部是可扩展的，因为新的首部（可以单独定义，从而提高模块化能力）可以通过下一个首部字段的方式在IP分组中链接在一起。因此，在IPv6首部的定义中存在“基本的”一词。

当然，主要的变化是将IP地址的大小提高为原来的4倍，从32位提高到128位。除了更多的可寻址空间的明显好处之外，这还可以支持寻址的层次结构。IPv6通过支持组播地址范围的定义，来提高组播路由的可扩展性。IPv6还正式定义了任播地址的概念。

IPv4生存时间（TTL）字段被重命名为跳数限制；服务类型字段被重命名为流量等级。其中，后一个字段支持服务质量，新的名为流标签的字段也是如此。接下来将再次回到这个问题上来，并对其进行详细的介绍。但现在值得一提的是，该字段支持的功能实际上等同于提供虚电路的作用。

定义基本IPv6首部的RFC 2460将流标签的功能描述如下：“一项新添加的功能，可以为发送者请求特殊处理的、属于特定业务‘流’的分组标上标签，例如非默认的服务质量或‘实时服务’”。增加这项功能的主要原因是为了纠正IPv4解决方案中存在的违背分层规则的问题，稍后将回顾这一问题。

其余 IPv6 首部规范的内容（基本首部之外的内容）请参阅 RFC 2460[1]。最后一点，尽管 IPv6 尚未完全部署，但 IPv6 网络以及支持 IPv6 的设备已经出现。IPv6 网络可以以隧道的方法通过 IPv4 实现互连。毕竟，开发 IP 的主要目标是互联网络。

到目前为止，只是简单地提到路由协议，其工作是计算路由映射。如图 4.14 所示。首先，LAN 中不需要路由，即使每个主机使用 IP（LAN 可以通过第 1 层和第 2 层交换机做进一步互连，构建更大的称为城域网的 LAN）。还可以使用电路交换技术（例如，租用电话线路）或虚拟电路交换技术［例如，帧中继或异步传输模式（Asynchronous Transfer Mode，ATM）技术，或者多协议标签交换（Multi－Protocol Label Switching，MPLS），稍后将讨论］将它们组织成广域网。然而，路由器加入时，它们需要了解网络中的其他路由器。

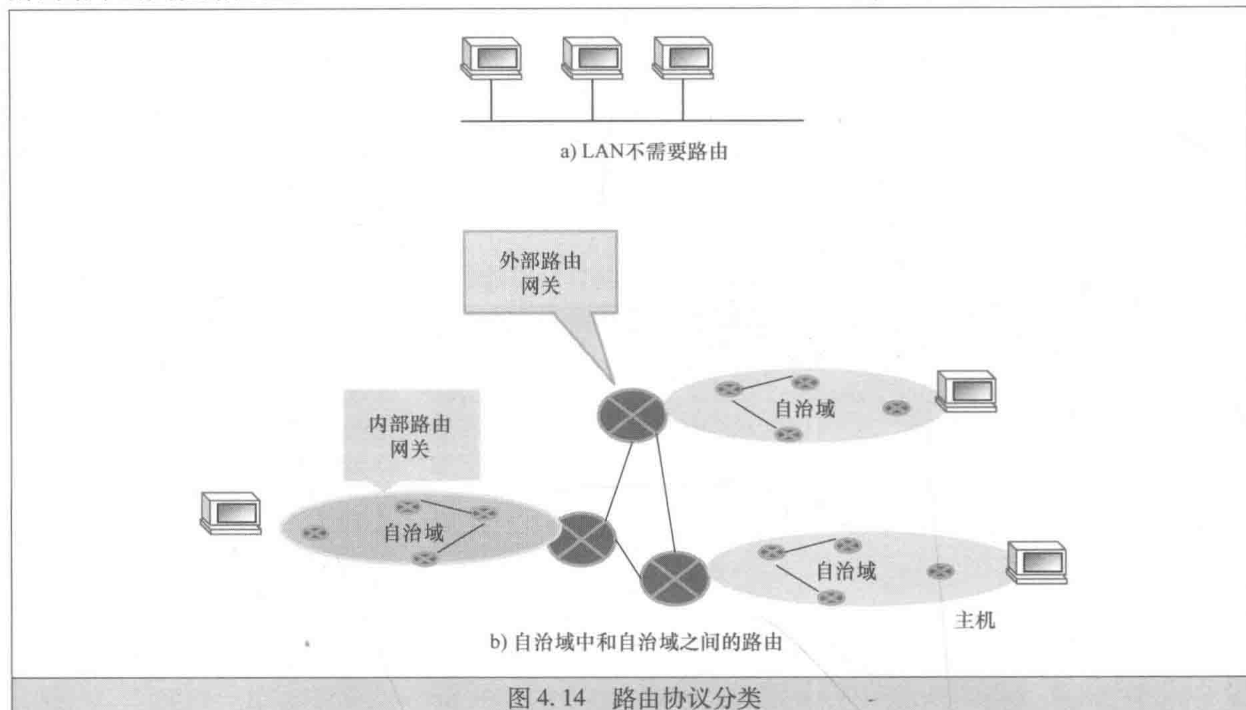


图 4.14　路由协议分类

问题是："路由到哪一个网络？"当然，没有路由器可以容纳整个互联网的路由映射。仅使用少量路由器的小型网络可以在其中提供静态路由映射。较大网络中的路由器需要实现这样或那样的路由网关协议。本书内容有限，无法对这些内容做进一步的论述[2]。然而，自治域（Autonomous Systems，AS）之间的外部路由的情况对于云计算尤其重要，因为它涉及提供"数据管道"服务。这里的想法类似于地理地图的开发：一幅国家地图，显示连接城市（外部路由）间的高速公路，还有一类地图显示城市中的街道（内部路由）。

如图 4.15 所示，每个 AS 由其各自的区域互联网注册机构（Regional Internet Registries，RIR）[3]分配一个称为自治域号（Autonomous System Number，ASN）的号码。对于世界其他地方，AS 由称为边界网关的路由器表示，交换由边界网关协议（Border Gateway Protocol，BGP）定义。BGP（当前版本号为 4）在 RFC 1771 中规定。

[1] 完整的 IPv6 必须包括/支持以下首部：逐跳选项、路由（类型 0）、分片、目的地选项、认证和封装安全有效载荷（Encapsulating Security Payload，ESP）。

[2] 这里推荐读者参阅本章参考文献［1］。

[3] RIR 为不同的地理位置分配了 IANA 颁发的块。RIR 将其管辖的范围按照字母顺序排列如下：非洲网络信息中心（African Network Information Centre，AfriNIC）（非洲）、美国互联网号码注册机构（American Registry for Internet Numbers，ARIN）［美国、加拿大和部分加勒比海地区（不包括 LACNIC 涵盖的地区）和南极洲］、亚太网络信息中心（Asia－Pacific Network Information Centre，APNIC）（亚太地区、澳大利亚、新西兰和该地区的岛屿）、拉丁美洲和加勒比网络信息中心（Latin America and Caribbean Network Information Centre，LACNIC）（拉丁美洲和部分加勒比海地区）以及欧洲网络资源协调中心（Réseaux IP Européens Network Coordination Centre，RIPE NCC）。



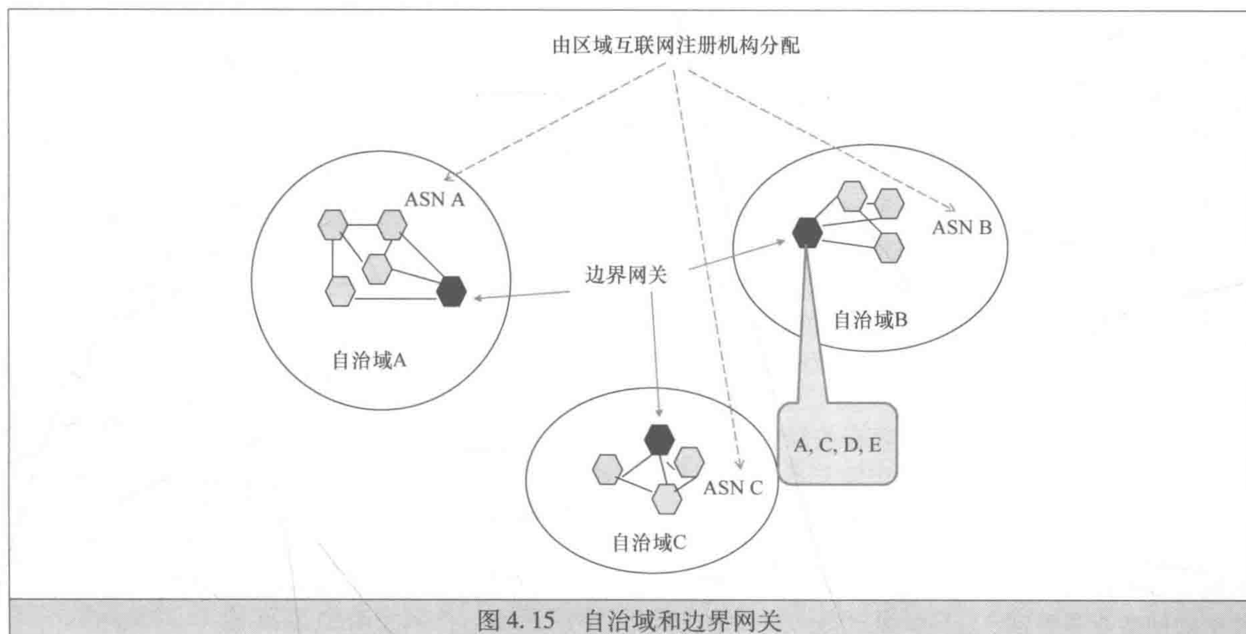


图 4.15 自治域和边界网关

过去，一个“自治域”意味着一个互联网服务提供商（Internet Service Provider，ISP），但是随着时间的推移，这种情况已经有所改变。现在，ISP 可以有几个单独的 AS 在里面。根据 RFC 1930“自治域的经典定义是在单个技术管理下的一组路由器，使用内部网关协议和共用的标准在 AS 中路由分组，并使用外部网关协议将分组路由到其他 AS。”

单个技术管理下的网络已经增长，否则演变成使用多个内部网关协议和多个标准。只要满足两个条件，同样的质量就仍然存在：①对其他网络来说，此类网络看似只有一个一致的内部路由计划；②通过这种一致的内部路由计划，可以识别哪些网络是可达的。后一个因素是不寻常的，因为不是每个网络都可以允许来自所有其他网络的流量流经它，相反，给定的网络可能不希望其流量通过某些网络。稍后，将回到这部分的内容上来。

考虑到这些变化，RFC 1930 将 AS 重新定义为“由一个或多个网络运营商运行的一个或多个 IP 前缀的连接组”。因此，该组必须有“一个单一且明确定义的路由策略”。这里的“策略”是关键，反映了内部和外部路由目标之间的主要区别。在内部路由中，目标是计算网络的完整映射，同时确定到每个网元的最有效的路由。

因此，当两个内部路由器连接时，它们交换关于它们知道的所有可达节点的信息（然后，基于该信息，每个路由器重新计算自己的网络映射）；相反，当路由器丢失连接时，它会重新计算它的映射并将更改通告给它其余的邻居。在这两种情况下，接收新信息的路由器会计算各自的映射，并将信息传播给它们余下的邻居，等等。

外部路由不是这样，它们会根据策略和协议来决定传播哪些信息。回到图 4.15。图中，自治域 B 知道如何到达自治域 A、C、D 和 E。既没必要、也不期望 B 自动地将其知道的所有的网络信息通告给 C。例如，为了让 B 向 E 和 C 通报：

1）E 的策略不得将 C 从它的中转目的地排除。

2）B 必须同意路由从 E 到 C 的流量。

3）C 必须同意接收来自 E 的流量。

4）E 必须同意接收来自 C 的流量（有对称性）。

事实上，AS 之间的关系分类法，如图 4.16 所示。

这里，开始进入业务领域的内容。应该注意到互联网是不受管制的，并且在很大程度上是自组织的。它已经并正在由业务来塑造。与普遍观点相反，互联网从来都不是免费的。在早些时候，它主要由美国政府支付费用，这恰好解释了图 4.10 的平面结构。自从那时以后，很多情况已经发生了改变。

在商业链的底部，ISP 客户向他们各自的 ISP 支付互联网连接的费用。

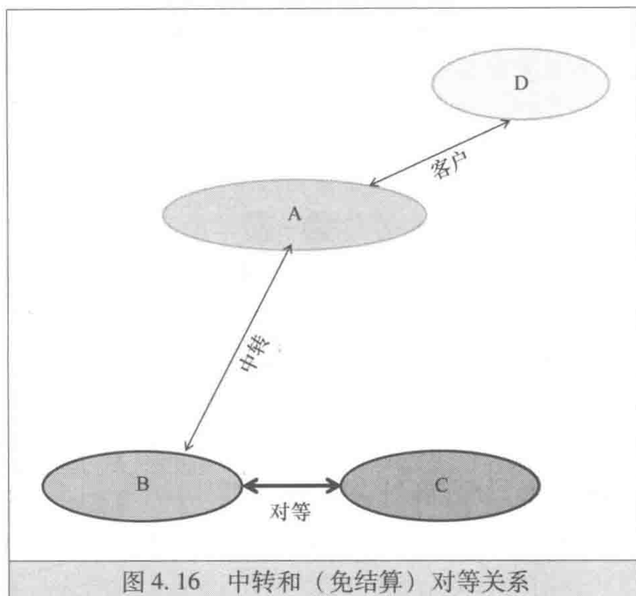


图 4.16　中转和（免结算）对等关系

较小的 ISP 通过为中转关系支付费用来实现互联互通，在这种中转关系中，结合了两种服务：①将路由通告给其他人（这具有向 ISP 客户征询流量或业务的效果）；②了解其他 ISP 的路由，以便向它们传递来自 ISP 客户的流量。流量在上游计费，也就是说，中转网络对其接收的流量进行计费，并为离开它的流量支付费用。

在对等体系中，只能交换网络与其下游客户之间的流量，并且两个网络都不能在对等连接上看到上游路由。一些网络是中转免费的，并且依赖于对等连接。

最初，“对等”一词意味着有关双方不需要向彼此支付任何费用，这是与中转关系不同的主要区别。后来，这个术语受到影响，在对等连接中出现了一个结算的概念。Geoff Huston［在澳洲电信（Telstra）时］有一篇文章[11]，描述了其中的细微业务差别㊀。

最终，最初的对等概念已经由“免结算”一词重新设定。通过由一级（Tier－1）网络构成免结算对等方式，大型的服务提供商可以实现完全的互联网访问。

两个网络可以通过专用交换机进行互连，但是当一个地区有两个以上的服务提供商时，通过独立运行的互联网交换点（Internet Exchange Point，IXP）互连对等网络更具有有效性。在撰写本书（英文原版）时，已经有超过 300 个 IXP。

看一下一级网络提供商的具体策略，会对理解这方面的内容更有启发式的帮助。

AT&T 已经发布了相应的文件，称为“AT&T 全球 IP 网络免结算对等策略”。这份文件（看的是 2012 年 10 月的官方版本）首次列出了公司的目标“在免结算的基础上将其 IP 网络与其他互联网骨干提供商互连起来，这样的互连可以为 AT&T 及其客户带来切实的好处。”之后，它提供了相关的 ASN：AS7018，用于在美国的专用对等；AS2685，用在加拿大，以及 AS2686，用在拉丁美洲。ISP 的对等请求必须以书面的形式提交，其中提供 ISP 服务国家的信息，包括已有的 IXP、服务的 ASN 和前缀集以及流量描述。

AS7018 规定了对等（在美国）的具体要求。首先，对等运营商必须具有“美国范围的 IP 骨干，其链路主要是 OC192（10Gbit/s）或更高”，并且在美国至少有 3 个点与 AT&T 互连，一个在东部海岸，一个在中部地区，一个在西部海岸。此外，候选对等体必须用两个“位于不同大洲（这里的对等具有重要的骨干网络）的非美国对等”与 AT&T 互连。AS7018 的客户可能不是免结算对等体。

其中对带宽和流量要求给出了明确说明：“到/从 AS7018 的对等流量必须按净计算，在美国到/从 AT&T 的主导方向在每月的繁忙时段必须达到平均至少 7Gbit/s”。因此，“在每个美国的互连点上，互连带宽必须至少为 10Gbit/s”。进 AT&T/出 AT&T 的流量比率被限制为不超过 2∶1。

对等的好处之一是合作解决安全事故和其他的运维问题。为此，候选对等方预计将通过“专业管理的 24×7［网络运营中心］NOC”参与其中合作，并用其来支撑参与这项合作所需的能力。

最后，必须考虑路由策略要求，这是我们最感兴趣的，因为它们说明了外部路由需要遵从的约束和限制：

1）对等方必须在每个互连点公布一套一致的路由。

2）对等方可以不公布中转或第三方路由，只提供它自己的路由和其客户的路由。因此，必须使用前缀来过滤对等方客户的路由。

3）禁止的活动包括“指定默认的路由……或以其他的方式转发未明确通告的目的地的流量，重置

㊀ 这篇文章发表一年后，2000 年 6 月，文章的作者参加了 Geoff 在互联网协会会议上的讲话，在那里他审视了 ISP 业务的财务结构并得出结论，认为这是一个面临危机的商品业务。后来人们将这个危机称为“互联网泡沫的破裂”。

下一跳，将下一跳出售或赠予其他人”。

考虑到业务焦点和产生的策略，BGP 是用于交换 AS 可达性信息的协议。执行 BGP 可以构建一个 AS 连接的有向无环图（即无环路），这代表了所有涉及 AS 的组合策略。为了确保这一点，RFC 1771 要求“BGP 发言者要向相邻 AS 中的对等体通告其本身所使用的那些路由”。这就不可避免地会导致“逐跳”路由的出现，而这在策略方面存在局限性。一个限制是不能执行源路由（即详细说明部分或全部的路由路径）。然而，BGP 确实支持所有符合“逐跳”范式的策略。

与其他路由协议不同，BGP 需要可靠的传输。乍一看可能很奇怪，路由协议实际上是一个应用层协议，但 BGP 是。路由是网络层提供服务的应用程序（类似地，网络管理应用服务于网络层，因为它们和其他层是一样的）。边界网关不一定需要在链路层互连，因此需要实现分片、重传、确认和排序等所有可靠传输层的功能。另一个要求是传输协议支持优安全关闭，确保在连接关闭之前，传送所有未完成的数据。或许不足为奇的是，本章前面介绍的 TCP[㊀] 已经满足了这些要求，因此 BGP 使用的是 TCP 传输。为此，TCP 的 179 端口被保留用于 BGP 连接。

现在，开始讨论传输协议，并继续讨论 IP 套件中的其他协议。

4.2.2 互联网沙漏 ★★★

图 4.17 的比喻出自于 Steve Deering（当时是互联网架构委员的成员之一），他于2001 年8 月8 日在伦敦举行的第 51 届 IETF 全体会议上，介绍了网际协议方面的进一步发展情况。

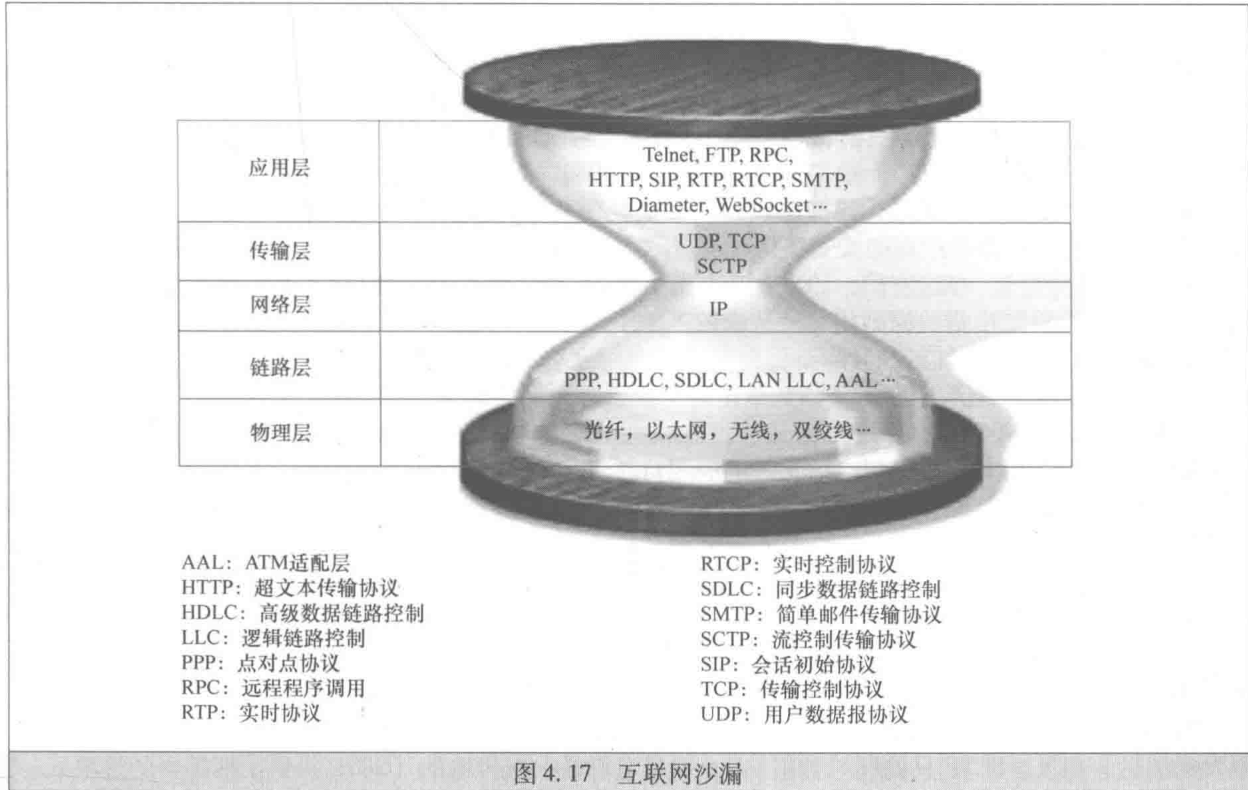


图 4.17 互联网沙漏

人到中年的 Deering 博士注意到，互联网发展了这么多年，已经步入了瓶颈期，而这个瓶颈就是 IP。链路层的协议很多，每一个都对应着相应的物理媒介，运行在 IP 之下，甚至还有更多的协议运行在 IP 以上，在传输层，特别是应用层。然而，IP 曾经是一个协议，也是一个非常简单的协议。互联网

㊀ 这里需要注意的是 TCP 和最初的传输控制程序愿景之间的差异，其中当前的 IP 和 TCP 被组合在一起。

㊁ 在 1992 年的一次会议上，穿着三件套西装的 Vint Cerf 脱下了他的外套、领带和白色的衬衫，露出了里面的带哥特式印花的 T 恤，他在会上指出：“一切基于 IP”。5 年后，在与世通公司（MCI Worldcom）总部的朗讯科技公司（Lucent Technologies）研究人员的会议上，Vint Cerf 穿着与5 年前那次会议同样的 T 恤，向当时的朗讯 CTO Bob Martin 介绍了相关的内容。Martin 博士问 Vint Cerf 是否这件 T 恤可以帮助他打到出租车，Cerf 博士答到：“只要司机关心的不是他车内的脚垫”。

的工作原理如 Vint Cerf 所认为的，主要体现为两点：IP 基于一切和一切基于 IP[⊖]。

在本节的其余部分中，将简要介绍部分网际协议族的内容，仅仅是为了开放视野。接下来将重新回到这些协议，并在本书后面更为详细地讨论这些协议。

物理媒介涉及有线和无线 LAN、同轴电缆和光纤链路，以及长距离无线广播与卫星通信。在诸如双绞线的点对点线路上，有 IETF 的点对点（Point - to - Point，PPP）协议和 ISO 的高级数据链路控制（High - Level Data Link Control，HDLC）协议。在 IBM 系统网络架构（Systems Network Architecture，SNA）的部署中，另一个被支持的第二层协议是同步数据链路控制（Synchronous Data Link Control，SDLC）协议——HDLC 以及几乎所有其他数据链路控制协议的前身。LAN 总是采用 IEEE 标准的逻辑链路控制（Logical Link Control，LLC）协议族，它们也携带了 SDLS 的基因。

ATM 网络是一个特例。正如在本章前面提到的，ATM 网络被定位为第 3 层网络。2001 年，尽管 IP 几乎赢得了这场争斗，但 ATM 交换机是 IP 路由器对等体的概念，仍然被部分行业所接受和分享。因此，有了 Deering 博士对网络层发展瓶颈的抱怨。在一个戏剧性的发展中，ATM 最终被归入到链路层，就 IP 而言：ATM 适配层（ATM Adaptation Layer，AAL）协议在沙漏模型中被视为第 2 层协议。IP 基于一切！

为此，IP 也可以在 IP 上运行。将 IP 分组封装到另一个 IP 分组（换句话说，通过附加新的 IP 首部，让一个 IP 分组成为另一个分组的有效载荷）的技术是完全合法的，并且已经被有效地用于创建一种虚拟专用网络。Deering 博士亲切地将 IP 的这个属性比喻成腰部，并在他的演示文稿中将该沙漏的腰部描述成扭曲的螺旋形。

在传输层有 3 种协议，全部由 IETF 制定。除 TCP 之外，它有效地组合了会话层和传输层业务，保证了端到端顺序、无错误的字节流传输，以及检测网络拥塞的特殊机制（并通过指数方式降低发送的数据量来对其进行调整），还有另外两种协议：用户数据报协议（User Datagram Protocol，UDP）和流控制传输协议（Stream Control Transmission Protocol，SCTP）。

RFC 768 中规定的 UDP 是无连接协议，不提供任何传送保证。UDP 也不需要任何连接建立。UDP 在实现由应用层执行差错控制的快速事务中是不可缺少的。另一个 UDP 的核心功能是传输“流媒体”（即语音或视频分组）。在这方面，UDP 更有优势，因为偶尔丢失的视频帧对视频服务的感知影响不大，而通过重传“挽救”出错的帧则相对会造成较多的不利影响（这会导致延迟方面的显著变化）。

后面将对 SCTP 进行详细的介绍，因为稍后需要用到它。需要弄清楚的是，为什么互联网会出现两种不同的可靠传输层协议。首先，SCTP 是作为本章参考文献［3］描述的 IETF Internet/PSTN 互通项目的一部分而制定的。SCTP 的初始目标是通过互联网传输 7 号信令系统消息，所以这项工作属于 IETF SIGTRAN 工作组。后来，一些与电话无关的应用协议规定 SCTP 作为选择的传输机制，主要是因为其内在的可靠性。该协议在 RFC 4960 中定义，并且通常情况下，还有一些其他相关的 RFC 也对其给出了相关的规范。

与 TCP 一样，SCTP 可以提供无差错的、不重复的有效载荷传输，采用数据分片和网络拥塞避免机制。SCTP 还解决了以下 TCP 中存在的限制：

1）TCP 将可靠传输与顺序传送结合在一起，但是在 20 世纪末，人们需要将这两个功能分离开，而 SCTP 允许应用程序从这两个功能中选择一个或两个都选择。

2）TCP 将有效载荷视为字节流。这迫使应用程序使用推送工具，在应用程序定义的记录结束时强制发送消息。相反，SCTP 只处理“数据块”，并且它们是一次传输的（SCTP 提供了捆绑块的选项）。

3）TCP 在多宿主（即主机被连接到多个网络上）方面是无效的，而 SCTP 明确支持这一功能，对于此稍后将予以展示。

4）TCP 一直存在容易受到同步拒绝服务攻击的漏洞。而 SCTP 通过在一种特殊的数据结构（cookie）中维护初始化状态，解决了这个问题。

图 4.18 通过一个例子对多宿主进行了说明。这里，进程 X 与进程 Y 之间有一个 SCTP 会话。与 TCP 的情况一样，SCTP 为每个进程均提供单个端口（进程 X 的 p_X 和进程 Y 的 p_Y），作为传输层服务的接口。

图 4.18 的例子与以前的情况非常不同的地方在于，运行进程的相应主机是多宿主的。运行进程 X 的主机连接到 3 个网络 A、B 和 C；运行进程 Y 的主机连接到两个网络 D 和 E。因此，前一个主机有 3



个 IP 地址（I_A、I_B和 I_C）；后一个主机有两个 IP 地址（I_D和 I_E）。SCTP 所添加的独特新功能是，在所有涉及的网络上，结合从这些网络传送到进程分组的多路复用，对进程接收的这些分组流进行去多路复用处理。此功能提高了性能和可靠性。当可用网络中有一个拥塞时考虑采用这一功能，其效果对性能的提升是比较明显的。

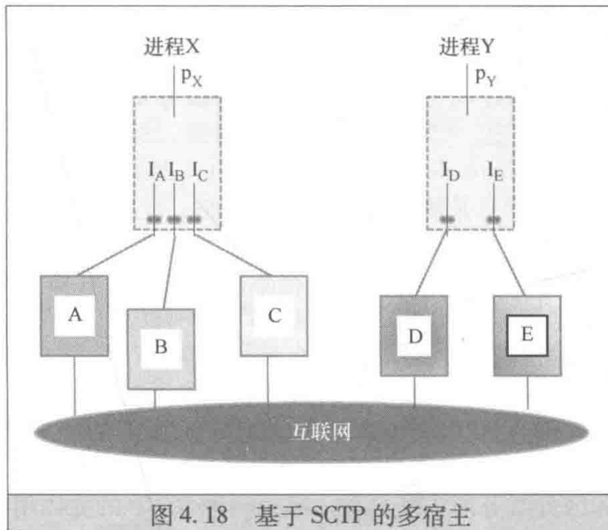


图 4.18　基于 SCTP 的多宿主

SCTP 对会话的含义进行了略微修改，将它称之为“关联”。一个 TCP 会话是一个由源 IP 地址、源端口号、目的 IP 地址和目的端口号定义的四元组（I_S，P_S，I_D，P_D），在 SCTP 关联中（$<I_S>$，P_S，$<I_D>$，P_D），参数 $<I_S>$ 和 $<I_D>$ 分别是源地址和目的地址的列表。

总而言之，人们注意到，在主要的操作系统中已经实现了 SCTP，并且已经被 3GPP 规定作为一种可选的传输层协议，专门适用于具有高可靠性需求的各种应用协议（例如，Diameter，一种用于认证、授权和计费的协议）。

在应用层协议方面，一个比较小的具有代表性的子集如图 4.17 所示。这些协议是针对特定需求制定的：用于电子邮件的简单邮件传输协议（Simple Mail Transfer Protocol，SMTP）、通过虚拟终端用于连接大型机账户的 Telnet、用于文件传输的文件传输协议（File Transfer Protocol，FTP）、用于对网元和设备进行远程管理的简单网络管理协议（Simple Network Management Protocol，SNMP）等。

超文本传输协议（Hyper - Text Transfer Protocol，HTTP）不仅定义并且开启了万维网（World - Wide Web，WWW），而且如稍后将看到的，它还影响了通用风格访问资源的创建及对其进行远程操作调用的方式。这种风格已经成为云计算应用程序访问的核心。HTTP 的局限性在于，只有 HTTP 客户端才可以开启与服务器的对话。其中，服务器只对客户端做出回应。对于全双工客户端到服务器通道，WebSocket 协议和 API 分别由 IETF 和 W3C 所开发和标准化。

10 年前，类似的接收服务器通知的功能被构建成 IETF 的会话初始协议（Session Initiation Protocol，SIP），制定该协议用于对多媒体会话进行创建和管理。作为第三代和第四代无线网络（以及固网的下一代网络）基础的 3GPP IP 多媒体子系统（IP Multimedia Subsystem，IMS）就是基于 SIP 构建的。

实际的实时媒体传输由 IETF 的实时协议（Real Time Protocol，RTP）执行。为此，（最初未预见到的）支持实时通信的需求已经促使了 QoS 发展，这部分内容将在下一节中介绍。在 Steve Deering 的演讲中，他把这个发展称为是具有“重量级”的，并在他接下来的演示文稿中介绍了一个腰部更加宽厚的沙漏。

4.3　IP 网络中的服务质量

看起来有些神秘的服务质量（Quality of Service，QoS）术语，实际上指的是一些相当具体且可以衡量的东西。在我们看来，这些“东西”可以归结成一些参数，例如带宽、延迟和延迟变化（也称为抖动）。遵守这些参数的某些值可以打造音频或视频服务。

在数据通信的早期阶段，几乎没有对延迟或延迟变化敏感的实时应用程序。电话（即使是数字化）仅由电话公司控制，而且它依赖于面向连接的传输，这提供了恒定的比特率传输。20 世纪 80 年代，综合业务数字网（Integrated Services Digital Network，ISDN，相关历史发展请参见本章参考文献［3］）电话公司的愿景是让话音业务流向传统交换，而数据业务被卸载到分组网络。随着提供带宽的增长，宽带 ISDN 计划设想了使用面向连接的 ATM 交换机。为此，有关 QoS 配置和保证的详细研究和标准化工作率先在 ATM 中展开，如本章参考文献［5］所述。

在 20 世纪 90 年代进行的支持 IP 网络的 QoS 标准化与 ISDN 无关，但它却是具有革命性的。术语“综合业务分组网”由 Jonathan Turner 创造，他在文章[12]中写道：“在本文中，我认为目前 ISDN 提案中

固有的演进方法不太可能提供有效的长期解决方案，并且我主张采用更先进的分组交换技术的革命性方法。本文的大部分内容都是针对综合业务分组网（Integrated Services Packet Network，ISPN）的详细描述，我建议将其作为当前 ISDN 提案的替代方案。”本章参考文献［13］对 Jonathan Turner 的这一愿景提供了进一步的理论支持。最终，综合业务模式的标准——特别是资源保留配置协议（Resource ReSerVation Setup Protocol，RSVP），由 IETF 在 20 世纪 90 年代末制定。这种综合业务模式有效地建立了（尽管只是暂时的）一种虚电路，使用路由器保持它的状态。

果然，ISDN 并没有成为“一种有效的长期解决方案”，但就在标准制定的时候，ISPN 革命也遇到了困难，稍后在讨论这项技术的时候，将解释这个问题。其对立面是区分服务运动（该名称本身是“综合服务”的双关语），1998 年 3 月接替了 IETF 的相关工作。区分服务模式暗示没有虚电路。路由器不保持状态，而是根据各自的类别来对待并处理到达的分组，如 IP 分组本身的编码。

最终，人们发现综合业务模式是无法扩展的，而且没有被部署。区分服务模式赢得了胜利；然而，这是因为基于虚电路的全新的网元技术的出现，多协议标签交换（Multi - Protocol Label Switching，MPLS）技术本身就将互联网和 ATM 方法综合起来了，因此赢得了最终的胜利。基于 MPLS，综合服务和区分服务模式结合在一起：一种 RSVP 的变体用于建立电路；区分服务用来维持电路上保证的 QoS。

本节的其余部分首先介绍了流量模型和 QoS 参数，然后解释了综合服务与区分服务模式和协议的内容。本节内容最终以对 MPLS 的介绍来结束全节内容。

4.3.1 分组调度规则和流量规范模型 ★★★

正如高速公路上行驶的汽车保持恒定的速度一样，通过网线传播的比特数据也是一样。在道路汇合或交叉的地方，汽车必须减速或停止，这时交通速度会发生变化；同样，网络流量速度变化发生在路由器的“交叉点”处。

路由器需要对到达的分组进行检查，以确定它需要向分组的目的地转发的接口（即地址），由于路由器中的 I/O 处理会消耗一定的时间，因此这自然会增加延迟。鉴于分组在几个接口上同时到达，很容易看到路由器如何成为影响性能的瓶颈。这些分组最终进入这样或那样的队列中，等待转发。只要排队分组的长度相加达到被分配路由器的内存限额，路由器内的分组才会被延迟，但是当这个限额达到时，路由器需要丢弃这些分组。

路由器接收分组时，可以根据这些分组各自的类型，对这些接收的分组进行处理，从而来塑造它所接收的业务流量，在根据分组类型对分组进行差异化处理的过程中，路由器采用的是分组调度规则，该规则可以选择分组传输的顺序（目前，特意使用抽象的属于类型，而不指定该类型是如何确定的。将在后续章节中介绍每个特殊情况下的分组分类）。

图 4.19 说明了 3 个主要的分组调度规则：

1）尽力而为，以先到先得的方式发送分组；当队列变得太大时，它们被丢弃。

2）公平排队，根据每个分组类型对分组进行排队，通过传输轮转，来保证每种类型都拥有相等的带宽份额。这里，在统计上具有更多分组的业务流量类型与具有较少分组的业务流量类型相比，后者将率先传送完毕。

3）加权公平排队，也采用轮转调度，但根据每种类型的优先级来分配带宽。与公平排队的区别在于，取代了从每个队列中传输相同数量的 n 个字节，而是根据队列中分组的类型 t 从每个队列中传输 nw_t 个字节。优先级越高的队列，分配的权重 w_t 越大。

事实证明（在本章参考文献［14］中证明），可以保证每种类型的端到端延迟和缓冲大小的上限。

当足够的路由器达到它们的极限时，整个网络就会拥塞。这种情况在运输和人群控制方面是众所周知的。在这两种情况下，通常在拥挤城市或拥挤地方入口采用准入控制策略。

网络接入中使用的两种流量规范模型基于更为简单的类比（见图 4.20），即底部有一个小孔的桶。这个孔象征着网络的入口。当桶满时，水开始溢出，不会进入到“网络”。

现在设想一个扮演塞子角色的主机，将分组插入到访问控制“桶”中，它是一个大小为 β 字节（桶深）的网络控制队列，以 ρ 字节/s 的速率来处理队列。

使用漏桶模型，一旦桶满了，以高于 ρ 的速率到达并导致溢出的分组将被丢弃。该模型消除了流量突发——流量可以以恒定的速率进入网络，而不能比这个恒定速率快。



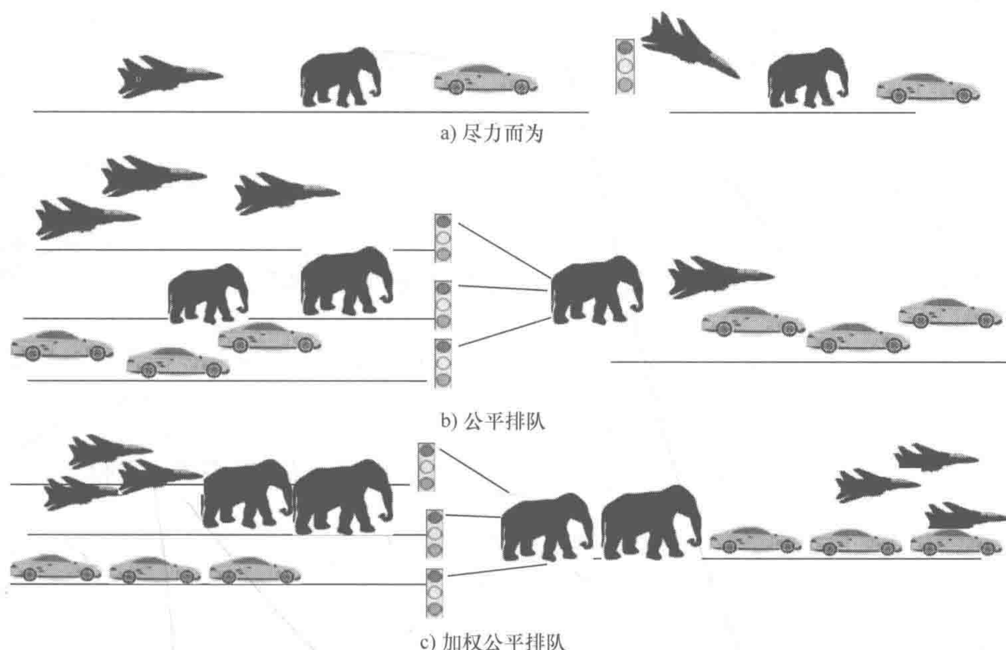


图 4.19　分组调度规则

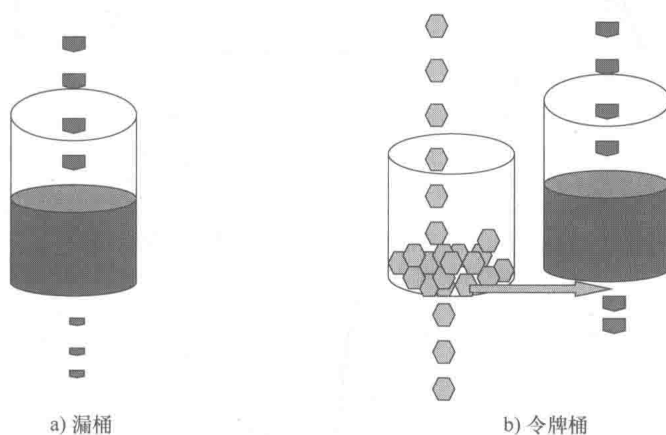


图 4.20　流量规范模型

令牌桶模型被设计允许突发。这里，从桶中流出的水流由阀门控制。阀门的状态由令牌桶的条件或者状况决定。令牌桶以 r 令牌/s 的速率接收称为令牌的量，并且它可以容纳 b 个令牌（当桶满时，令牌停止到达）。现在，当离开令牌桶时，令牌会打开主桶阀门［见图 4.20b］，来允许输出一个字节，此时，阀门关闭。因此，当令牌桶为空时，不允许产生流量。然而，如果没有流量输出，令牌桶底部的孔将会关闭，因此，令牌被保存在桶中直到它们开始溢出。令牌桶与漏桶模型的区别在于，现在可以允许达到令牌桶大小的突发产生。

因此，t 时间段内准许流量的容量 $V(t)$ 被限定为

$$V(t) \leqslant rt + b$$

因此，最大的突发 M 可以持续（$M - b$）$/r$ 秒。

通过在强制速率为 R 的漏桶后面放置一个令牌桶，可以使流量不会超过该速率，同时允许受控的突发存在。

4.3.2 综合服务 ★★★

在综合服务模型中区分分组的类型由流定义。流是一个五元组：（源 IP 地址，源端口，协议，目的端口，目的 IP 地址）。其中，协议用于指定传输协议。在定义该模型时，只有两个这样的协议：TCP 和 UDP（SCTP 是在后来定义的，而且由于它的多路复用特性，并不完全符合，除非扩展流的定义允许将源 IP 地址和目的 IP 地址扩展成 IP 地址组）。需要注意的是，分层原则在这里被打破，因为路由器必须检查 IP 有效载荷。

在最简单的情况下，流是由网络层向传输层提供的单工（即单向）端到端的虚电路；然而，综合服务使用了组播方式，因此在一般情况下，这里的虚电路是一棵“树”，它的“根”源于要发送分组的点。

综合服务框架是为了补充 IP 而精心构建的，但为了支持它，IP 路由器必须在很大程度上进行改变。这种改变包括进行预留的必要性，即建立电路，然后在需要时保持这些电路。

RFC 1633 中规定的 IETF 1994 综合服务框架定义了一种新的路由器功能，称为流量控制，采用3个组件来实现：准入控制、分类器和分组调度器（顺便提一句，区分服务框架在很大程度上重用这些术语）。

准入控制的作用是决定一个新的流是否可以获得所需的 QoS 支持。具有相同类型的多个流接受相同的 QoS 处理是可能的。然后，分类器将传入的分组映射到它们各自的类型（也称为服务类）上。最后，分组调度器在每个转发接口上管理按类排列的分组队列，并在转发接口上串行化产生分组流。

上述实体在路由器中，也在端点主机上建立和设置。由 RSVP 启动综合服务，RSVP 在端点主机和之间的路由器中，为每个流创建和维护一个针对每个流的特定状态。稍后将详细讨论 RSVP，但通常的想法是，接收应用端点根据流规范或 flowspec 指定 QoS 参数，然后穿过网络送到源主机[⊖]。在通过路由器的过程中，flowspec 需要在每个路由器中通过准入控制检查；如果检查通过，则预留被接受，预留设置代理更新路由器流量控制数据库的状态。

图 4.21 展示了对 RFC 1631 框架图的一些修改，解释了上述要素之间的互通性。控制面和数据面之间有明显的区分，在控制面，信令与实际数据传输异步发生；在数据面，分组实时流动。有趣的是，RFC 1631 的作者设想由网络管理来完成预留，即不使用路由到路由的协议。路由器的这种功能与转发无关。

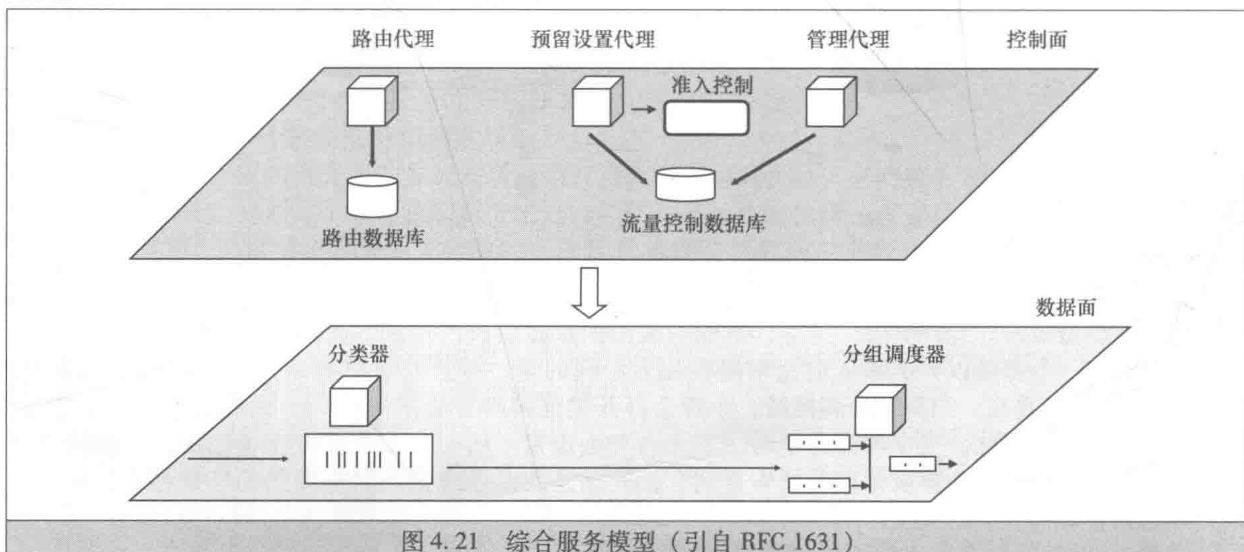


图 4.21 综合服务模型（引自 RFC 1631）

综合服务模型涉及两种端到端服务：按流保证服务和控制负载服务。

保证服务为符合条件的流提供保证的带宽和端到端排队延迟的上限。一致性由流量描述符（Traffic

⊖ 如果这看起来有些奇怪，可以考虑一下 20 世纪 90 年代的流媒体视频应用程序。大概这种接收终端应该订阅服务（甚至可以为其所需的 QoS 支付网络费用），因此对于它来说开启预留是合理的。

descriptor，TSpec）定义，这是服务对网络的义务。网络对应用程序的义务在服务规范（Service Specification，RSpec）中定义。

TSpec 基于令牌桶模型，包含了 5 个参数：

1）令牌速率 r（字节/s）；

2）峰值速率 p（字节/s）；

3）令牌桶深度 b（字节）；

4）最小监管单元 m（字节）（如果分组较小，仍将其计为 m 个字节）；

5）最大分组大小 M。

RSpec 包含了两个参数：服务速率 R（字节/s）和为了在调度中引入一些灵活性的“放松条件” S（μs），它是可以在仍然满足端到端延迟上限时添加的延迟时间。

图 4.22 提供了关于 TSpec 和 RSpec 参数保证服务在最糟糕情况下延迟时间的公式。这里有两个附加的路由相关变量：

C_i：由于分组长度和传输速率，分组在路由器 i 中经历的开销。

D_i：由于流识别、流水线等，分组在路由器 i 中经历的与速率无关的延迟。

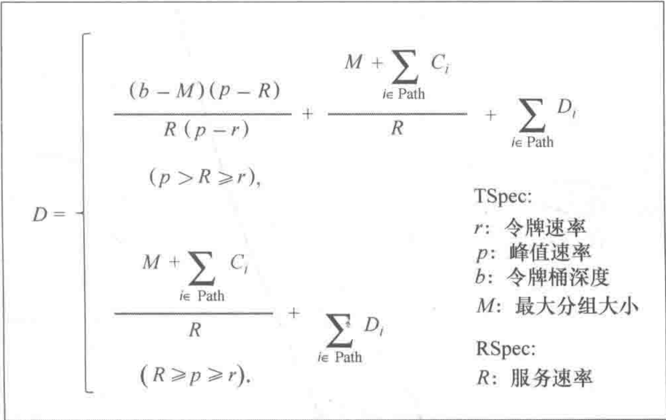


图 4.22 端到端最糟延迟 D（引自 RFC 2212）

与保证服务不同，控制负载服务最好根据不允许发生的情况进行描述，即可见的排队延迟或可见的拥塞损失，这样便于理解。这个定义是相当含糊的，没有定量的保证，因为准入控制留待实现（有时，这项服务被称为更好的尽力而为的服务）。使用这项服务，路由器依赖于统计机制，避免了昂贵的预留。因此，控制负载服务只需要 TSpec（其中限制了应用程序可以插入到网络的流量），而不需要 RSpec（其中规定了网络义务）。

1997 年，IETF 完成了综合服务标准，并在 RFC 2205 到 RFC 2216 的一系列 RFC 中为此目的[⊖]指定了 RSVP。

图 4.23 提供了使用 RSVP 的简单示例。综合服务接收方的主机使用 RESV 消息启动预留。此消息携带流描述符、时间值（刷新所需，将在稍后解释）、目的 IP 和端口地址、协议号等参数。请求通过路由器上行传播，直到它们到达提供这项服务的主机。

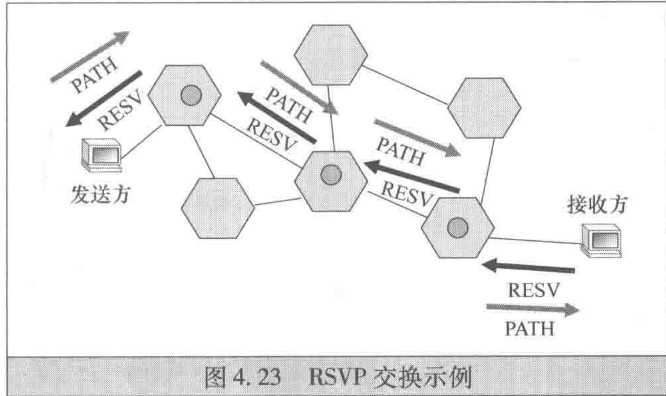


图 4.23 RSVP 交换示例

后一个主机使用 PATH 消息进行响应，PATH 消息下行传播，并在路由器中安装状态。该消息包含流标识、发送方 TSpec、时间值和其他参数（顺便提一下，出于安全性的考虑，RESV 和 PATH 消息都包含了其完整性的加密证明，即证明它们没有被篡改）。作为 PATH 消息传播的结果，所有路由器都安装了预留状态（图 4.23 中用小圆圈表示）。接受“在网络中保持状态”，意味着早期互联网原则的重大妥协；然而，这种状态被声明为“软”状态。只要 RESV 和 PATH 消息在指定的时间段内总是到达，它就会被保持，并且实际上程序在会话期间总是在不断地发布它们。这应该就解释了两种消息中需要时间值的原因。如果交换停止，则状态被销毁。

会话也可以使用 PATHTear 消息拆除。图 4.24 中的表格提供了完整的 RSVP 消息集。

稍后将会看到，其他模型中 RSVP 的作用将与综合服务中的作用不同。

⊖ 如前所述，RSVP 扩展已被用于不同的控制面目的，这远远超出了综合业务的范围。

4.3.3 区分服务 ★★★

称为区分服务，是为了强调与综合服务的区别。正如提到的那样，这些服务又是以 ISDN 命名的（因此，这里的词源仍然指向电话网络）。然而，这两种模式中的服务本身几乎相同，只是使用它们的手段不同。制定新模式的主要原因是，综合服务（其预留和状态机制）不会在大型核心网络中扩展。

消息	方向
PATH	下行(发送方到接收方)
RESV	上行(接收方到发送方)
PATHerr	上行(响应PATH)
RESVErr	下行(响应RESV)
PATHTear	下行
RESVTear	上行
RESVConf	下行(响应RESV中的特定请求)

图 4.24　RSVP 消息总结

与综合服务模式一样，服务（就 QoS 而言）以其端到端的行为为特点。但是这里没有定义服务。相反，该模式支持一组预先定义好的独立于服务的模块。类似地，转发处理的特征在于每个路由器的行为（分组调度），并且这种模式定义了转发行为（而不是端到端行为）。

另一个巨大的差异是，没有保留 QoS 的信令协议。因此，在路由器中不会保留状态。此外，在这种模式中，没有流的意识或任何其他虚电路的结构。分组所属的流量类型（根据它进行处理）被编码在分组的 IP 首部字段（IPv4 中的服务类型，IPv6 中的服务等级）中。这些类型被称为类别，路由资源被分配给每个类别。

代替信令，网络配置被用于向路由器提供必要的参数。定义好的转发处理可以组合起来提供新的服务。

类别处理基于客户和服务提供商之间的服务水平协议（Service - Level Agreements，SLA）。流量在网络边缘得以被监管；之后，网络本身就采用基于类别的转发。

下面看一些细节。每个每跳行为（Per - Hop - Behavior，PHB）被分配一个 6 比特的区分服务编码点（Differentiated Services Codepoint，DSCP）。PHB 是构建模块：具有相同编码点的所有分组都会进行行为聚合，并获得相同的转发处理。PHB 可以进一步组合成 PHB 组。

在区分服务（Differentiated Service，DS）域内的所有路由器中，PHB 的处理都是相同的。再次，出于提供 QoS 的目的，源地址和目的地址、协议 ID 和端口是无关紧要的，只有 DSCP 是重要的。

客户和服务提供商之间以及两个相邻领域之间的服务水平协议（SLA）规定了这些服务。SLA 规定了流量以及安全性、计费、账单和其他服务参数。SLA 的核心部分是流量控制约定（Traffic Conditioning Agreement，TCA），其定义了流量规格和相应的监管措施。这些示例是每个类别的令牌桶参数（吞吐量、延迟和丢弃优先级），以及不符合条件分组的操作。

图 4.25 说明了这些概念。SLA 可以是静态的（即提供一次）或动态的（即通过实时网络管理动作来改变）。

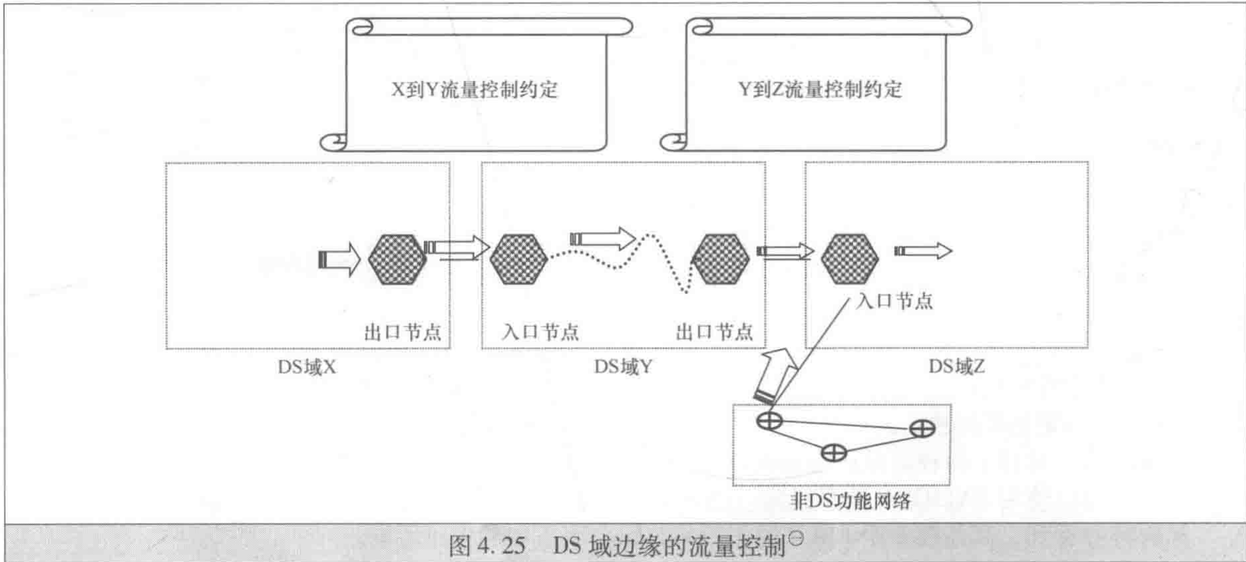


图 4.25　DS 域边缘的流量控制[⊖]

⊖ 经美国阿尔卡特朗讯公司许可转载自本章参考文献［3］。

就 PHB 分类而言，默认的 PHB 组对应于好的旧的尽力而为的处理。实际的从类别选择器（Class Selector，CS）PHB 组（按升序优先级顺序列举的 CS－1～CS－8）开始。

下一组是快速转发（Expedited Forwarding，EF）PHB 组，保证会聚分组的离开速率不低于到达速率，其中假设 EF 流量可以抢占其他流量。

保证转发（Assured Forwarding，AF）PHB 组将优先级（按升序排列）分配给 4 个服务类别，并为每个服务类别定义了 3 种丢弃优先级（作为规格外分组的处理）。

图 4.26 为保证转发提供了一个不言而喻的说明：针对每个运输类别，列出它的优先级以及丢弃优先级。



图 4.26　AF 规范示例

图 4.27 给出了路由器内部这种模式操作的解释。

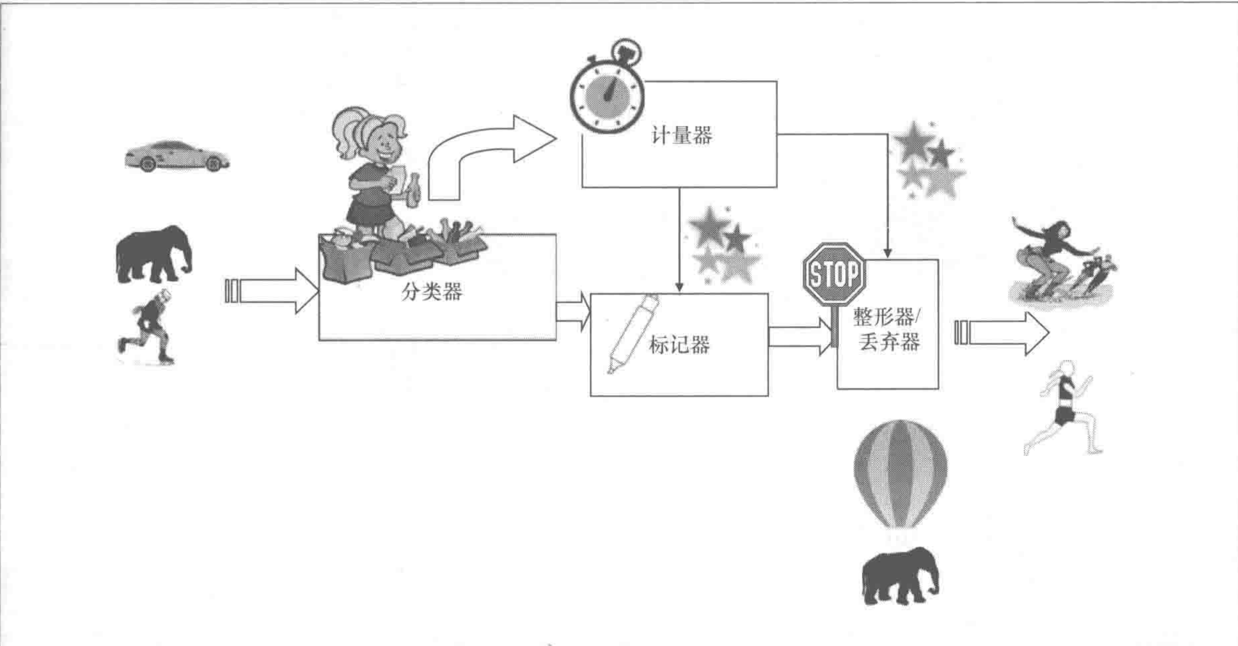


图 4.27　DS 的内部视角（引自 RFC 2475）

这种架构的主要要素包括分类器、计量器、标记器、丢弃器和整形器。

分类器的功能在网络边缘和内部节点上是不同的。在位于网络边缘的边界节点上，分类器确定分组所属的聚合和相应的 SLA；在网络中的其他地方，分类器仅检查 DSCP 值。

计量器根据流量协议规范（Traffic Agreement Specification，TAS）检查聚合（传入分组所属的聚合），并确定其是否在类别规格的范围中。根据特定的情况，分组被标记或被丢弃。

标记器在 DS 字段中写入相应的 DSCP。标记可以由主机完成，但由边界节点检查（并且在必要时可能会被改变）。在某些情况下，可能会使用特殊的 DSCP 来标记不符合条件的分组。如果发生拥塞，这些被判定的分组可能会被丢弃。根据 SLA 约定的流量规格，分组也可能被升级或降级。

整形器延迟不符合条件的分组，直到它带来相应的聚合符合流量规格。为此，通常需要在出口节点和入口节点之间进行整形。

总而言之，以上给出了关于各自标准相应的说明。在不同的时期，总共有十几个区分服务 RFC 被 IETF 发布。RFC 2474 将 PHB 映射到 IP 分组的 DS 编码点。RFC 2998 利用了将边缘网络中的综合服务和核心网络中的区分服务结合在一起的想法。鉴于网络管理在这里的特殊重要性，强烈建议读者阅读 RFC 3279 和 RFC 3280，它们分别定义了路由器中的管理信息库（即所有参数的集合），并解释了网络管理操作。

4.3.4 MPLS ★★★

Deering 在他的 IP 沙漏演讲中，提到了 MPLS 以及它所支持的多种技术，Deering 博士把它们称为“腰下凸起”，叹息道“糟糕的是，IP 已经做的（或可以做的，或应该做的）大部分是重新创造”。也许情况是这样，但是 IP 并没有提供一种简单的虚电路交换方案。这并不奇怪，因为这样的方案会与互联网的基本原则相抵触。

鉴于 20 世纪 90 年代中期，ATM 交换机以及帧中继交换机都需要做一些工作来综合面向连接和无连接的技术。本章参考文献［15］介绍了这种后来被称为“标签交换”的综合，其中毫不含糊地指出了，“通过采用通用的寻址、路由和管理程序，可以简化路由器和异步传输模式交换的集成”。

IETF MPLS 工作组自从 1997 年以来，一直处于活跃工作的状态。它已经成为最繁忙的 IETF 工作组之一，已经发布了 70 多份 RFC。

借助图 4.28 可以比较明显地看出路由和交换之间的区别，接下来在此基础上开始对 MPLS 的论述。

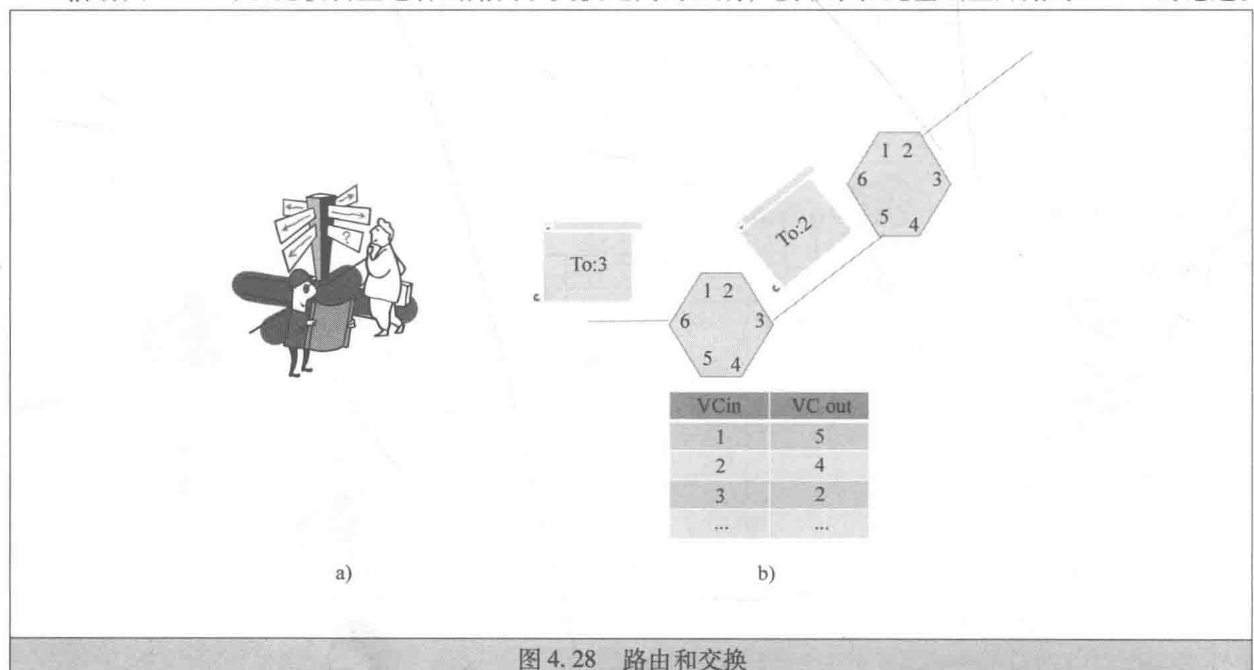


图 4.28 路由和交换

想像一下，在一个陌生的城市里，手里拿着地图朝着所给地址的地标出发。为了到达该地标，一个人需要明确知道此刻自己在哪里，找到这个地方和它在地图上的地标，找到一组看起来像到达该地



标最短路径的街道，然后开始往这个目标走，检查拐角处街道的名称，如图4.28a所示，决定向哪个方向转弯。这需要一段时间，而过时的地图可能会使事情进一步复杂化。这就是路由存在的问题。如果是走路的话，那么在这些情况下，每个开车的人都知道这会更让人崩溃（周围其他被挡着路的司机会不停地按喇叭！）

但是，当所有人都要做的就是跟着路牌标识向目标地标出发时，（无论是步行还是驾车）找到目的地这件事就变得非常简单了。这就是在预设电路上交换的情况。电路没必要是永久性的（就像标志着地标建筑物方向的永久路牌和在道路修复期间临时放置的绕行路牌）。还有一个比喻是美国高速公路的路线命名。从迈阿密到波士顿，只需要沿着95号州际公路即可，实际上这条公路延伸到几条不同的高速公路上，这些公路甚至以不同的方式命名。但是，交叉路口有很好的路牌标志作为指示，只要汽车司机跟着这些标志（而不是看地图），汽车就可以从一条高速快速切换到另一条高速公路上，而在这个过程中，车辆始终行驶在这条路线上。

类似地，使用数据报交换，网络节点不必通过确定数据报的目的地将其转发到下一个节点。相反，数据报提供了一个（本地有意义的）接口号——虚电路ID，使得数据报能够根据它命名的接口进行精确的转发。例如，图4.28b左侧的节点已经接收到要在接口3上转发的数据报。该节点的交换映射图表明下一个节点转发该数据报的接口号为2，所以虚电路ID 3被下一条虚电路ID 2所替代。因此，数据报跟着预设的“电路”进行转发。

这正是MPLS的工作原理。首先，标签交换路径（Label - Switched Path，LSP，一种虚拟的端到端电路）被建立；然后，这种交换机（也称为支持MPLS的路由器）交换（而不是路由）这些IP分组，而不会看到分组本身。而且，LSP可以沿着任何路径（不一定是最短的路径）建立，这种功能有助于流量工程。

本地有效电路由标签指定（因此，MPLS包含L）。如图4.29所示，MPLS标签在链路层帧内作为IP分组的前缀，就在原本是链路层首部的位置。在ISO参考模型方面，将MPLS简单放置在第二层使其并不实用，“真正”的链路层都是直接在物理电路上运行的。为此，MPLS有时被称为“2.5层技术”。由于标签可以堆叠，因此要想实现建立虚拟专用网的目的，这种情况将进一步复杂化（与虚拟化一样，需要引入递归）。为了简单起见，假设该堆栈只包含一个标签。

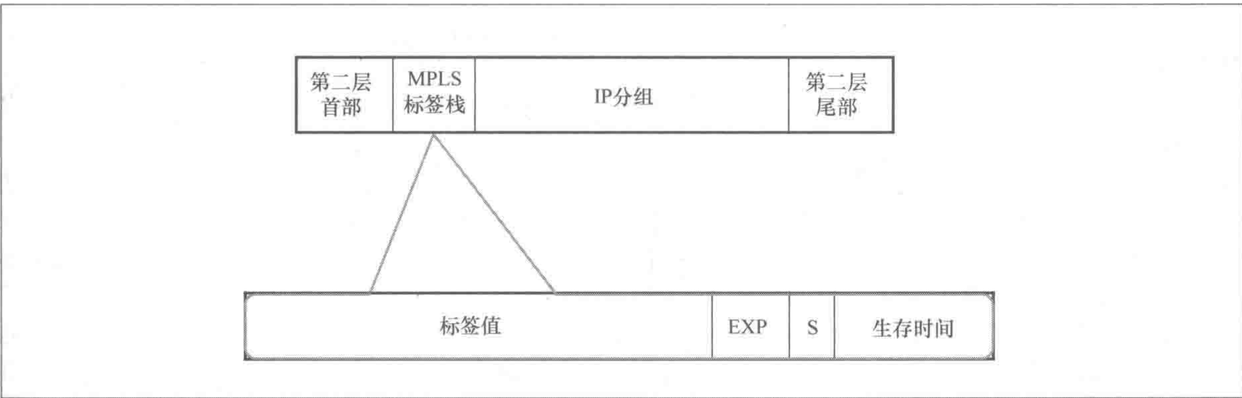


图4.29　MPLS标签位置和结构

MPLS标签的结构在RFC 3032中定义。标签包含4个字段：

1）标签值（20位），即实际的虚电路ID。

2）流量类别（Traffic Class，TC）（3位）[㊀]，用于与diffserv相互配合和显示拥塞标记。

3）S（1位），是一个布尔值，指示当前标签是否是标签栈上的最后一个。

4）生存时间（Time - to - Live，TTL）（8位），与IPv4中的同名字段具有相同的目的和相同的意义（MPLS不检查IP分组）。

㊀ 该字段最初称为EXP（表示“实验”），尽管原始计划是尝试对QoS类别进行编码。由于没有对这些事情达成一致，因此就产生了使用“实验”这一结果作为策略上的一种妥协。这种犹豫不决被证明是危险的，因为其他标准化组织将该字段的名称解释为邀请他们尝试使用他们自己的字段。考虑到即将失去对标准的控制，IETF重新命名了这一字段，这是通过RFC 5462实现并对外发布的。

标签的使用方式很简单。标签值作为 MPLS 交换机内部表［称为入标签映射（Incoming Label Map, ILM)］的索引，其中包含出标签、下一跳（即转发接口的索引）以及与路径相关联的状态信息。然后，形成新的链路层帧，将新的出标签插入到 IP 分组的前面。

这里有一个有趣的问题：在这个框架中，有效载荷必须是 IP 分组吗？答案是，这根本不是必须的。目前，正在 IP 网络中讨论 MPLS 的使用，自然需要从 IP 的角度来看待所有的内容，但是很快就会用到各种有效载荷，包括 ATM 信元和以太网帧，它们都可以使用 MPLS 转发。这最后解释了 MPLS 中的“多协议”部分。然而，MPLS 是 IP 网络的发明，其主流的用途还是在 IP 网络中。为了强调这一点，MPLS 标准将 MPLS 交换机称为标签交换路由器（Label - Switched Router，LSR)。

虽然交换的过程相当简单，但是工程化、建立和拆卸端到端电路的整体问题就变得非常复杂。业界（尤其是 IETF）多年来一直在解决这个问题，并且在它的很多其他方面开展工作，特别是光纤网络方面[⊖]。

图 4.30 说明了标签到 LSP 的映射及其一些重要的属性。首先，所有的 LSP（就像综合服务流一样）都是单工的。例子实际上只涉及流，并且只看其中 3 个：

1）流（a）源于主机 C，并终止于主机 B。为了负载平衡（一种重要的网络功能，在本书中将多次回到这个问题上)，有两条 LSP。LSP I 遍历了 3 个 LSR，即 S_1、S_2 和 S_4，最后两个分别分配流标签 20 和 30。类似地，LSP II 遍历了 3 个 LSR，即 S_1、S_3 和 S_4，将标签 72 和 18 分配给了同一个流。

2）流（b）源于主机 C，并终止于主机 D，也穿过了 LSR S_1、S_3 和 S_4，并为该流分配了标签 25 和 56。

3）流（c）源于主机 A，并终止于主机 D，穿过 LSR S_2 和 S_4，后一个交换机分配了流标签 71。

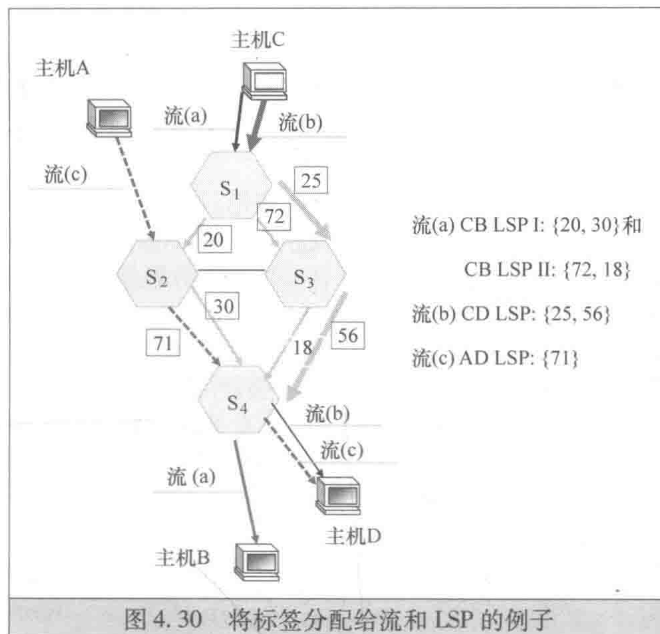


图 4.30　将标签分配给流和 LSP 的例子

标签交换路径（LSP）中的每个标签与转发等价类（Forwarding Equivalence Class，FEC）之间具有一对一的关联，后者反过来又与分组接收的处理相关联。FEC 由一组规则定义。例如，规则可以是，FEC 中的分组与匹配特定的 IP 目的地址，或者它们发往相同的出口路由器，或者它们属于特定的流（就像图 4.30 示例中的一样）。当然，不同的 FEC 具有不同的可扩展性。FEC 分组的分类由入口路由器执行。

在这一点上，我们准备简单讨论一下 LSP 的形成问题。这是一个简单的规则，因为只有特定的 LSR 才知道如何索引它自己的入标签映射，该规则指的是，向上（upstream）分配标签，即从信宿到信源与 RSVP 中进行预留的方式类似。

实际上，RSVP 在流量工程中的扩展被称为 RSVP - TE，已经成为标签分发的标准协议。因为，RSVP 最初设计用于支持组播，所以 RSVP - TE 的开发重新开启了对宽带组播的兴趣。

在 RSVP - TE 之前，人们还设计了一个协议用于标签分发，这个协议被称为标签分发协议（Label Distribution Protocol，LDP)。后来，LDP 的设计人员同样需要考虑流量工程的问题，因此产生了 LDP 扩展协议，该协议被称为基于约束的路由 LDP（Constraint - based Routing LDP，CR - LDP)。但是它仍然无法简化这个问题，因此 BGP 引入了自己的管理 LSP 的扩展协议。

⊖ 除了 IETF 之外，MPLS 的工作已经在几个行业论坛（例如，ATM 论坛和帧中继论坛）上就其他“多”部分的协议方面取得了进展。目前，这些论坛已经整合到了宽带论坛（BroadBand Forum，BBF）上。此外，一些 MPLS 的标准化工作也已经由 ITU - T 完成。参见本章参考文献［16］，可以了解到与 ITU - T 早期工作有关的分组交换网络中的 QoS 方面内容的概述。

图 4.31 给出了一个基于 CR - LDP 和 RSVP - TE 的简单设置路由的示例。该示例主要关注的是它们之间的相似之处，而不是它们之间的差异。请求流向上游，而标签的分配则在下游进行。除了建立 LSP 之外，两种协议都支持 LSP 隧道重新路由，应用“先通后断（make - before - break）”原则，并提供抢占选项。

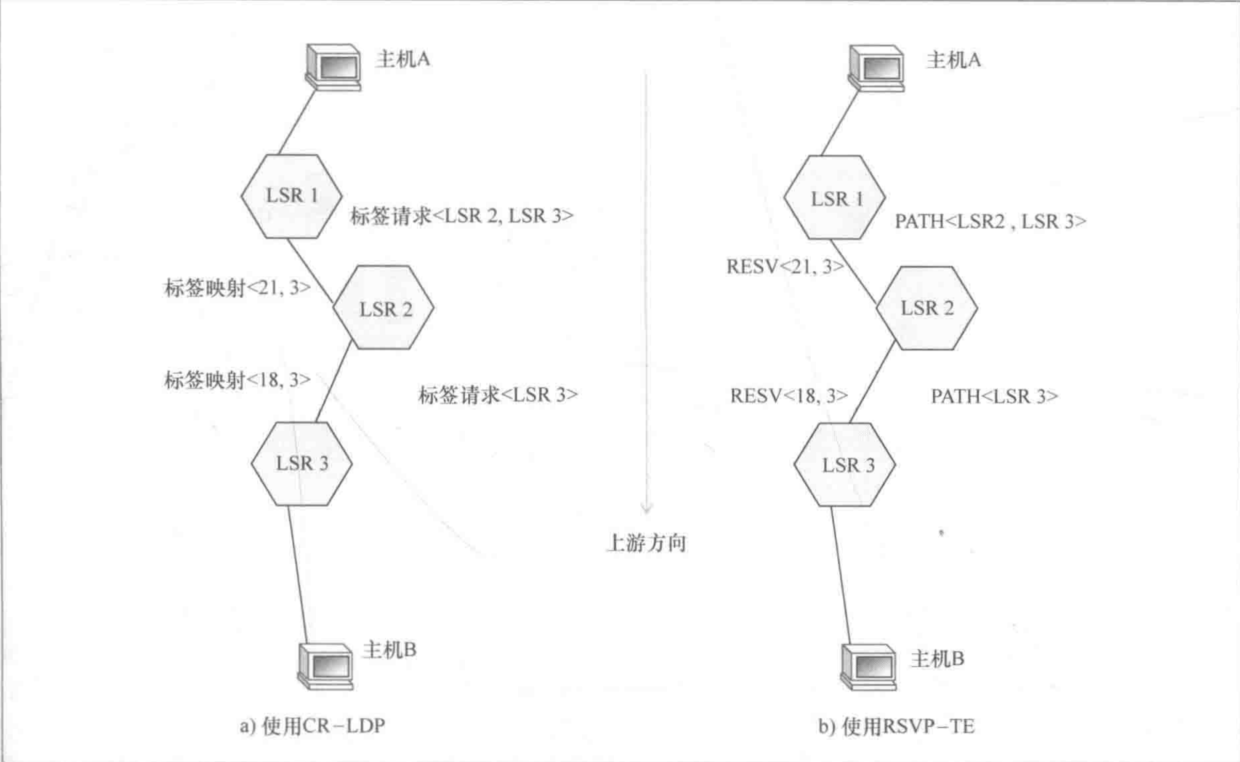


图 4.31　使用 CR - LDP 和 RSVP - TE 显示路由设置的例子

除了对组播的支持外（这是 RSVP - TE 的原有特性），CR - LDP 和 RSVP - TE 之间的差异是微不足道的。简而言之，这些都体现在底层协议中（CR - LDP 需要 TCP 或 UDP，而 RSVP - TE 直接在 IP 上运行）、安装在 LSP 中的状态［CR - LDP 安装硬状态，而 RSVP - TE（以其原始设计）安装软状态］；LSP 刷新（仅由 RSVP - TE 执行）和安全选项（RSVP - TE 有它自己的认证机制）。

总而言之，MPLS 出现以来不足 20 年，但已经走过了很长的路。首先，它能够与 ATM 和帧中继交换机进行端到端通信。它还引入了一种支持互联网流量工程的手段（例如，卸载、重新路由和负载均衡），加速分组转发，以及为虚拟专用网的建立提供一种连贯的原则（在下一节将会看到）。

MPLS 技术，通过其被称为广义 MPLS（Generalized MPLS，GMPLS）的扩展协议版本，也已经发展成可以支持其他交换技术（例如，那些涉及时分复用和波长交换的技术）。关于这方面的内容，有一本具有权威性的著作[17]。随着开始讨论 WAN 虚拟化技术（下一节的主题），需要注意的是，MPLS 已经被证明是这一领域中的首选技术。

4.4　WAN 虚拟化技术

正如在引言中提到的，虚拟数据网的需求早于 PDN 时代。事实上，PDN 的创建是为了解决企业需要拥有自己独立的专用网络的需求。

为此，所有的这些术语“虚拟化”意味着，在应用到数据网络时，只是将某些东西放在现有网络上的一种环境，作为一个覆盖层，实际上就是从其中切出非相交的、同质的专用网络部分。换句话说，这些部分中的每一个都具有由它对应的专用网络定义的寻址方案，并且它根据网络策略进行操作。

构成 VPN 的端点的关联可以在不同的 OSI 层上实现。这里考虑了第一层、第二层和第三层 VPN。

图 4.32 所示的第一层 VPN（Layer - 1 VPN，L1VPN）框架和标准已经由 IETF 和 ITU - T 指定。

RFC 4847 中对该框架进行了描述，其中还列出了相关文档。L1VPN 被定义为“由核心第一层网络提供的服务，用于在两个或多个客户站点之间提供第一层连接，并且其中客户可以对该连接的建立和类型有一定的控制”。该模型基于 GMPLS 控制的流量工程链路。因此，数据面（而不是控制面）是电路交换的。

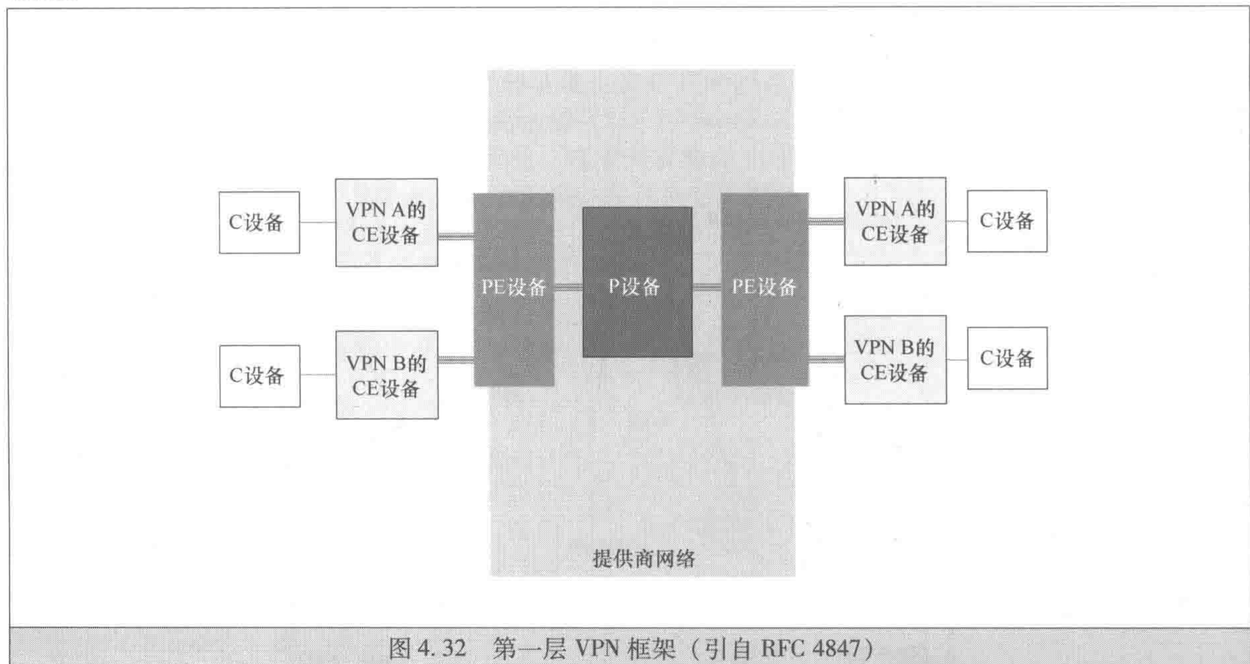


图 4.32　第一层 VPN 框架（引自 RFC 4847）

作为客户设备的 C 设备，在作为 VPN 端点的客户边缘（Customer Edge，CE）设备上聚合（图中有两个 VPN，名称为 A 和 B）。CE 设备可以是时分复用（Time Division Multiplexing，TDM）交换机，但它也可以是第二层交换机，甚至是路由器。定义的 CE 设备功能是“能够接收第一层信令，并进行交换或通过适配终止它”。

反过来，C 设备连接到提供商边缘（Provider Edge，PE）设备上，该设备是通过其分配 L1VPN 服务的提供商网络的互连点。PE 设备可以是 TDM 设备、光交叉连接交换机或以太网专线设备（通过 TDM 传输以太网帧）。PE 设备本身由交换机［称为提供商（Provider，P）设备］互连。

VPN 成员关系信息由这组 CE 到 PE GMPLS 链路定义。即使 CE 设备属于客户，它们的管理也可以外包给第三方，这就是第一层 VPN 的优点之一。另一个好处是“小规模”地使用传输网络：几个客户共享物理层基础设施，而不用投资建造它。

第二层 VPN 有两种。一种是简单的“伪电路”，它提供一种点对点的链路层服务。另一种是类 LAN 的，因为它提供了一种点对多点的服务，将多个 LAN 互连到 WAN 中。这两种都使用了相同的协议来实现它们的目标，所以将只讨论第二种［虚拟 VLAN（Virtual LAN，VLAN）］使用的这些协议。

首先，虚拟 LAN 有两个方面。第一个方面涉及将“真实的”（非模拟的）LAN 切割成看似独立的单独 VLAN。第二个方面涉及用胶合替代切割，处理的是链路层上的连接 LAN。

图 4.33 说明了 VLAN 的概念。执行不同功能的主机［图 4.33a 使用不同的形状描绘］共享相同的物理交换 LAN。目标是通过软件方式实现图 4.33b 所示的逻辑分组，其中每个功能都有自己的专用 LAN。为此，逻辑 LAN 甚至可以具有不同的物理特性，使得例如多媒体终端获得更高的带宽份额。

通过 VLAN 感知交换机的实现符合 IEEE 802.1Q 标准。交换机识别表征帧所述类别特点的标签，并相应地传送帧。

VLAN 可以通过伪电路（如 RFC 3985 中定义）的方式扩展到 WAN 第二层 VPN，其中在 LAN 的情况下，帧的比特流通过分组交换网传输。

图 4.34 中重用了图 4.32 中的术语，展示了这一框架。这里的区别是，第二层帧在模拟（而不是真实）电路上通过 PSN 进行隧道传输。

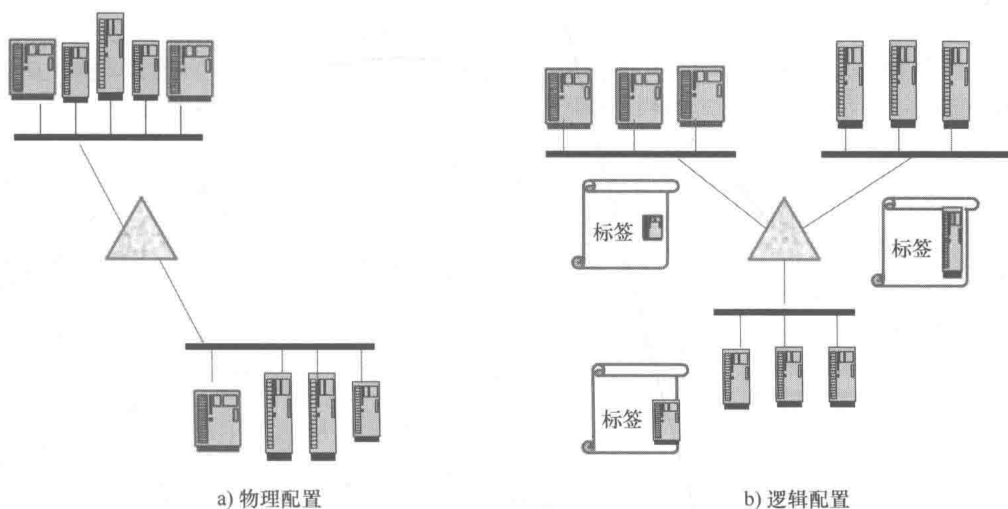


图 4.33　VLAN 概念

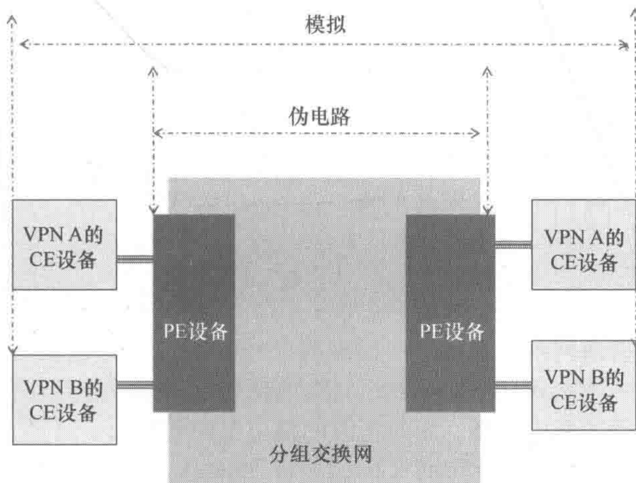


图 4.34　伪电路模拟边缘到边缘网络参考模型（引自 RFC 3985 中的图 2）

可以自然地在 MPLS 电路上建立伪电路，而不需要 MPLS 存在。存在一种在 IP 网络上以第二层隧道协议（Layer Two Tunneling Protocol，L2TP）的形式承载链路层分组的较旧（并在某种程度上具有竞争性的）技术。最初，如本章参考文献［3］所述，L2TP 是在支持 PSTN/互联网集成的第一种机制中制定的，可以通过电话线连接主机和互联网路由器。它旨在携带具有单个点对点连接的 PPP 帧。现在其第 3 版中，RFC 3931 中规定的 L2TPv3 仍然保留了呼叫建立术语，但其应用范围已经扩展到支持基于以太网的 VLAN。

最后，从概念上来看，第三层 VPN 没有太大的不同——也涉及隧道技术，但是在第三层上使用而已。在这种情况下，PE 不必处理 LAN 特定的问题，而是存在另一组处理重复地址的问题。每个第二层网络地址卡都有唯一的第二层地址；然而，IP 地址不是这样。稍后，当介绍网络地址转换（Network Address Translation，NAT）设备时，其原因将变得很清楚，但实际上两个专用网络拥有重叠的一组 IP 地址是可能的，因此提供商必须能够消除它们的歧义。

RFC 2547 描述了一组技术，其中 MPLS 被用于通过提供商的网络进行隧道传送，并且使用路由区分字符串扩展了 BGP 通告路由器，这可以消除同一个 PE 中重复地址的歧义。

另一种方法是 RFC 2917 中引入的虚拟路由器架构。这里，MPLS 也被用于隧道传输，但不需要专

门的路由通告机制。歧义消除是通过在不同的路由域中使用不同的标签来实现的。

到目前为止，MPLS 在构建 VPN 方面，尤其是在支持 QoS 的流量工程方面的作用和好处是显而易见的。现在，可以解释需要堆栈 MPLS 标签的原因了，这正是因为要满足 VPN 要求而提出的。

图 4.35a 描述了专用网络中通过 MPLS LSP 互连的两种不同的 LAN。现在考虑一下另一种使用 VPN 的互通方案，如图 4.35b 所示。当 LSP 进入入口 PE 时，后者可以推送自己的标签（保留原始标签，在提供商网络中没有意义，因此与有效载荷一起通过它进行隧道传输）。该标签在出口 PE 处弹出，因此该分组返回到带有原始标签的专用网络，该标签现在对其余 LSP 有意义。

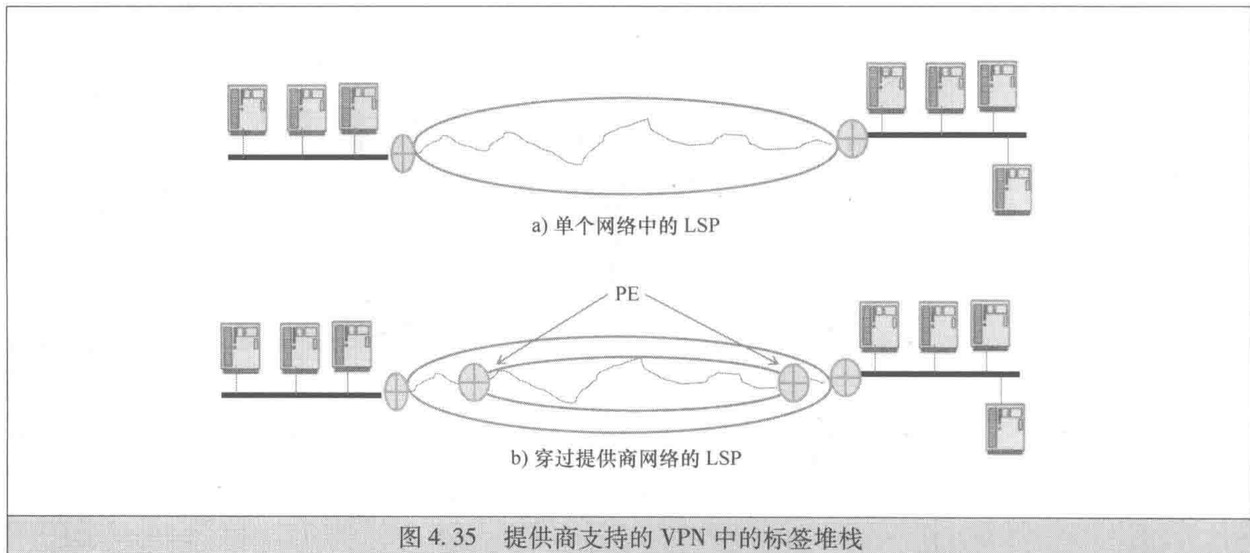


图 4.35　提供商支持的 VPN 中的标签堆栈

在讨论现代云数据中心时，将再次提及 VPN。

4.5　软件定义网络

SDN 的思想——集中管理网络，在网络一出现就已经存在了（可以认为，从人类的一开始它就已经出现了，因为它涉及集中计划和管理的好处，与本地自主管理的敏捷性和鲁棒性相对）。应当弄清楚的是，SDN 的思想不是一般的集中管理，而是实时的路由集中管理。

为此，最初 SDN 的发展与 PSTN 智能网（Intelligent Network，IN）的发展非常类似[18]。电话网络一直都是集中管理的，但电话交换机通过彼此之间的合作建立呼叫。在 20 世纪 70 年代，它们的发展使得呼叫建立通过单独的分组网络在语音话路的带外进行。到 20 世纪 90 年代，呼叫控制逐渐与全软件服务控制分离。后者开始于简单的地址转换（与互联网域名服务器执行的不同，有关这方面的内容，将在本书后面详细介绍），逐渐演变为执行复杂的服务逻辑程序。

该演进路线正在导致交换机成为简单的“交换架构”，所有其他功能都转移到通用的计算机中，从而对呼叫和服务逻辑进行完全控制，将 IN 和网络管理相结合。电信信息网络联盟（Telecommunications Information Networking Consortium，TINA - C）在这一愿景下已经研究了 7 年（从 1993 年到 2000 年），提供了架构、规范甚至软件。虽然方向是正确的，但这一愿景却从来都没有实现，仅仅是因为这种“交换架构”在互联网上已被取消——它的功能由 LAN 交换机[⊖]和 IP 路由器所承担。这就是电话交换机的开始。

在 20 世纪 90 年代后期，几个 IETF 工作组研究工作（详细描述在本章参考文献［3］）的进展导致电话交换机的明确标准被有效地分解成 PSTN 和 IP 部分。“交换”是 IP 路由器的功能。因此，软交换

⊖　事实上，从 20 世纪 80 年代贝尔实验室的实验可以看出，电话业务可以部署在以太网的 LAN 上，并不需要其他的“交换架构”。1994 年，以色列一家公司（Vocaltec）的 Alon Cohen 和 Lior Haramaty 专门为 IP 语音（Voice - over - IP，VoIP）收发器申请了专利应用。该专利于 1998 年获得。

的概念诞生了[⊖]，它的名字反映了其可编程性的想法，自然而然，只需要通用业务的计算机就可以来控制它。在 IETF 传输领域，由 LuHuilan 博士带领会话初始协议/智能网（Session Initiation Protocol/Intelligent Network，SIN）设计团队演示了如何通过软交换机重新使用和增强所有基于 PSTN 的业务。这项工作的结果在很大程度上被忽视，因为 3GPP 正在进行一项相关但更强大的工作——移动网络 IP 多媒体子系统（IP Multimedia Subsystem，IMS）的标准化，即 RFC 3976。在对 RFC 的贡献中，其主要作者 Vijay Gurbani 博士提供了其部分博士论文的研究成果。随着 IMS 的成功，软交换的名字逐渐被人们所淡忘，与软交换一起的还有业界对 PSTN 交换机的增强甚至维护方面所做的努力。

随着电话交换机的消失，研究人员的注意力开始转移到 IP 路由器。后者开始于通用计算机（例如，数字设备公司的 PDP-10 小型计算机，在 1977 年已经被 BBN 部署在 ARPANET 中），但到 2000 年，它们已经变成由专有操作系统控制的复杂的专用设备[⊜]。然后，这推动了将路由器的转发部分（需要专门硬件的“交换架构”）与执行信令和构建路由表的部分分开。

2000 年，IETF 开始讨论转发和控制元素分离（Forwarding and Control Element Sepa ration，ForCES）工作组章程，2004 年，ForCES 框架作为 RFC 3746 发布。图 4.36 再现了 ForCES 架构。

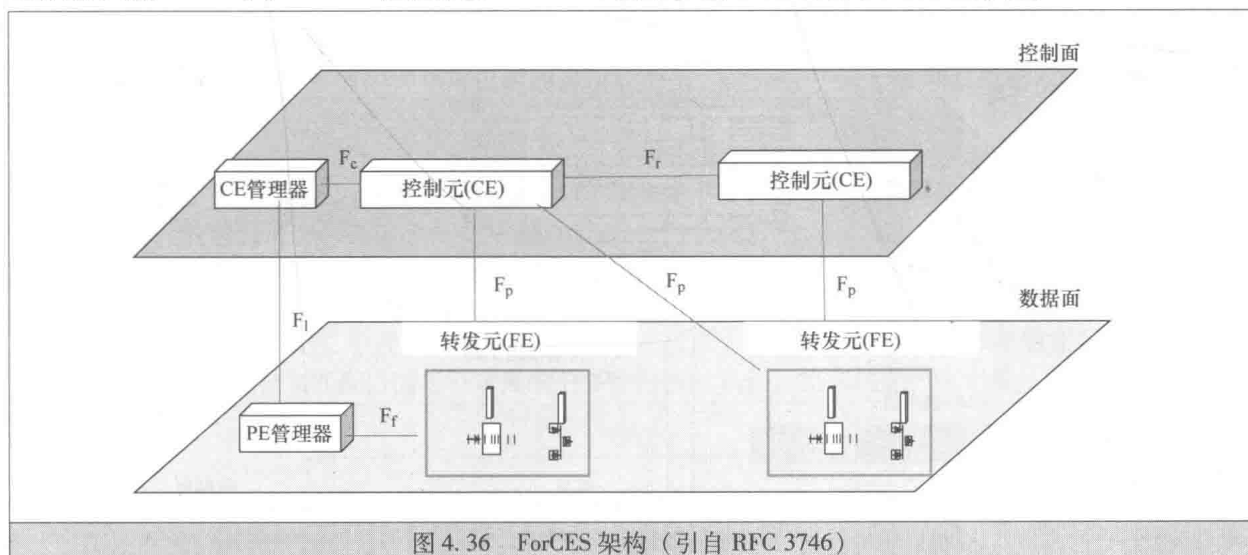


图 4.36 ForCES 架构（引自 RFC 3746）

RFC 5810 在一对网元之间定义了 ITU-T 传统中称为参考点的接口协议（或者更确切地说是多个协议）。面向事务的协议消息是为协议层（Protocol Layer，PL）定义的，而协议传输映射层（Protocol Transport Mapping Layer，ForCES TML）则“使用现有的传输层协议的功能来专门解决协议消息传输问题”。具体的 TML 在单独的 RFC 中定义。

到 2010 年为止，已经有了几个 ForCES 协议的实现，在 RFC 6053 中给出了其中三个的报告并对其进行了比较。在贝尔实验室的软路由器项目中，ForCES 的发展进入了一个新的高度。在本章参考文献［19］内描述的软路由器架构中，控制面功能完全与分组转发功能相分离。没有静态关联；FE 和 CE 相互动态发现。当 FE 启动时，它会发现一组可能控制它的 CE，并将其自己绑定到“最佳的”CE 上。这种对软交换看似间接地提及，实际上是一个直接的参考。正如本章参考文献［19］的作者所说：“所提出的网络演进与目前正在发生的基于软交换的语音网络架构转型有相似之处。引入软交换架构，用于将语音传输路径与呼叫控制软件相分离。软路由器架构旨在通过将转发元与控制元分离的方式在路由分组网络中提供类似的迁移。与软交换类似，软路由架构降低了向网络中添加新功能的复杂性”。

接下来的重要内容是反映人们普遍的学术观点：互联网架构日益“僵化”，不可能改变；实际上，当没有大规模访问真实网络进行实验的时候，很难通过单纯的讲授让学生们掌握实际网络方面的知识。

⊖ 当时，也是国际软交换联盟（International SoftSwitch Consortium，ISSC）的项目。

⊜ 复杂性是由不断增长的实时性要求驱动的。特别是，转发部分（即数据面）必须足够快，不过即使最快的总线架构也无法跟上需求的步伐。因此，许多快速路由器最终使用好的老式交叉开关交换（纵横式交换机），就像电话交换机一样。

全球网络创新环境（Global Environment for Network Innovations，GENI）项目已经通过建立一个可编程网络的广泛项目来解决了这个问题，可编程网络使用虚拟化技术（包括网络覆盖），以便允许“网络切片”，从而为单个研究人员提供看似 WAN 的实验环境。

2008 年，8 名研究人员将其概念缩小到校园网络，发布了他们所谓的“白皮书”[20]㊀。这篇文章的建议被称为 OpenFlow。

人们观察到，当前以太网交换机和路由器中的流表具有共同的信息，因此本章参考文献［20］的作者提出利用这一点，以便直接对流表进行编程。然后，系统管理员可以通过划分业务流量并将流表中的条目分配给不同的用户，来切片校园网络。这样一来，研究人员“就可以通过选择他们的分组所跟随的路由器及其接受的处理，来控制他们自己的流”。通过这种方式，研究人员就可以尝试新的路由协议、安全模型、寻址方案，甚至 IP 的替代方案。

根据本章参考文献［20］，理想的 OpenFlow 环境如图 4.37 所示，有意将该图绘制在前一个图的上面，以展示其发展的过程。这里，在通用计算机上实现的控制器进程，在交换机中与其对应的进程维护着一条安全的传输层信道。后一个进程负责维护流表。维护流表的 OpenFlow 协议在这两个进程之间进行交换。

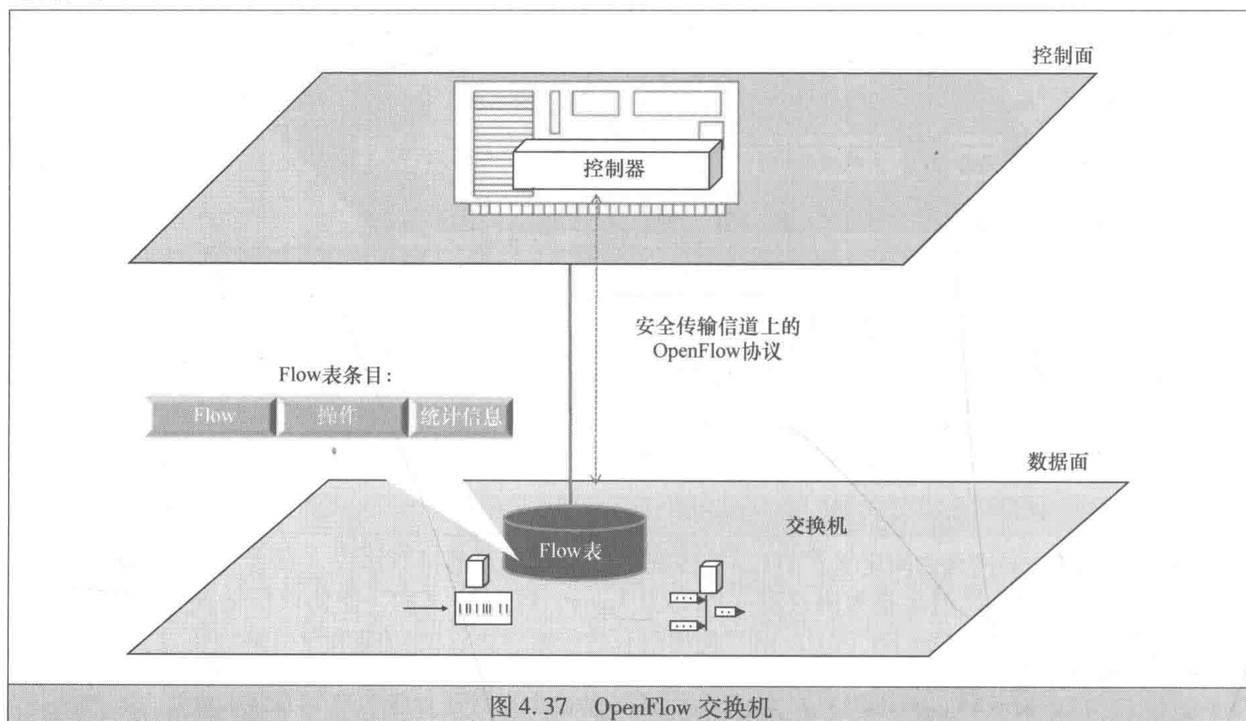


图 4.37　OpenFlow 交换机

流有着广泛的定义：它可以是共享相同五元组（如综合服务中定义的）或 VLAN 标签，或 MPLS 标签的分组流。或者它还可以是从给定的第二层地址或 IP 地址发出的分组流。除了定义流的首部之外，流表条目还定义了要对分组执行的操作，以及与每个流相关联的统计信息（分组数、字节数以及自定义流的最后一个分组到达的时间）。该操作可以是在特定接口上发送分组，也可以是将分组丢弃。或者，在这里，将分组发送给控制器，进行进一步检查。显然，这一功能对研究人员有利，并且其执法效果同样明显。

因此，行业进一步将数据网络架构转换成完全可编程的实体，即在 20 年的时间内，实现 IN 和 TINA－C 计划。2008 年，OpenFlow 联盟成立，旨在制定 OpenFlow 交换机规范。本章参考文献［20］报告称，该联盟的成员资格“面向全世界的学校、学院、大学或政府机构人士都是自由开放和免收费用的”。然而，该联盟为了消除对供应商的影响，限制了那些“非受雇于生产或出售以太网交换机、路由器或无线访问点公司”的个人成员。2011 年，这种情况发生了变化，该联盟的工作由开放网络基金会

㊀　该文是提交给 ACM 计算机通信评论的一篇社论，没有经过同行评审。



(Open Networking Foundation，ONF）所接管，该基金会拥有庞大且多样化的企业成员，包括网络运营商和供应商。其中有一个特别计划，用来为研究人员提供支持，为其提供免费的会员资格。

4.6 IP 安全性

网络安全有很多方面，本书将持续讨论这一问题，不断介绍新的内容。简单提一下后面几章涉及的此类相关内容，有防火墙（这是 IP 流量的“检查站”），还有网络地址转换（Network Address Translation，NAT）设备（它的目的是隐藏网络的内部结构，通常与防火墙组合使用）。然后，还有一些访问管理机制，以及不同的 OSI 各层加密机制。这里强烈推荐读者参阅本章参考文献［21］，它是一本内容全面的网络安全方面的专著。鉴于本章关注的主要是网络层方面的内容，因此本书将只讨论一个方面的内容：IP 安全性。

在最初设计 IP 时，不需要考虑安全性，所以 IP 分组从源地址到目的地址都是以明文的形式自由穿梭于互联网之上。只要互联网是面向研究人员的网络或研究人员使用的网络，那么这种情况就没有问题（至少他们认为是没有问题的）。

事实是，广播网络上的任何人（或访问交换机）都可以窥视分组中的内容（如果只是为了获知别人的密码）；具有更先进手段的人还可以更改或重放分组。此外，任何人都可以轻松地将任意具有伪造源 IP 地址的分组注入到网络中。

人们制定了称为 IP 安全（IP security，IPsec）的协议族，用于在网络层提供安全服务。在网络层设计安全性的一个重要原因是，现有的应用程序可以保持不变，而新的应用程序仍然可以在人们无须知道相关复杂性的情况下进行开发。在之下层次的安全性处理方面，只提供了一种逐跳解决方案，这是由于端到端路径中的路由器原因（对于传输层的安全性处理，最初被认为是多余的。并且如果网络没有被“破坏”，人们将仍然认为这样做是多余的，这在后面将表现得比较明显）。

IPsec 在一组 IETF RFC 中规定，其关系如图 4.38 所示。了解这种关系对 IPsec 的实际使用很重要，因为它提供了可以混合和匹配以满足特定需求的多种服务（认证、机密性保护、完整性保护和防重放）。

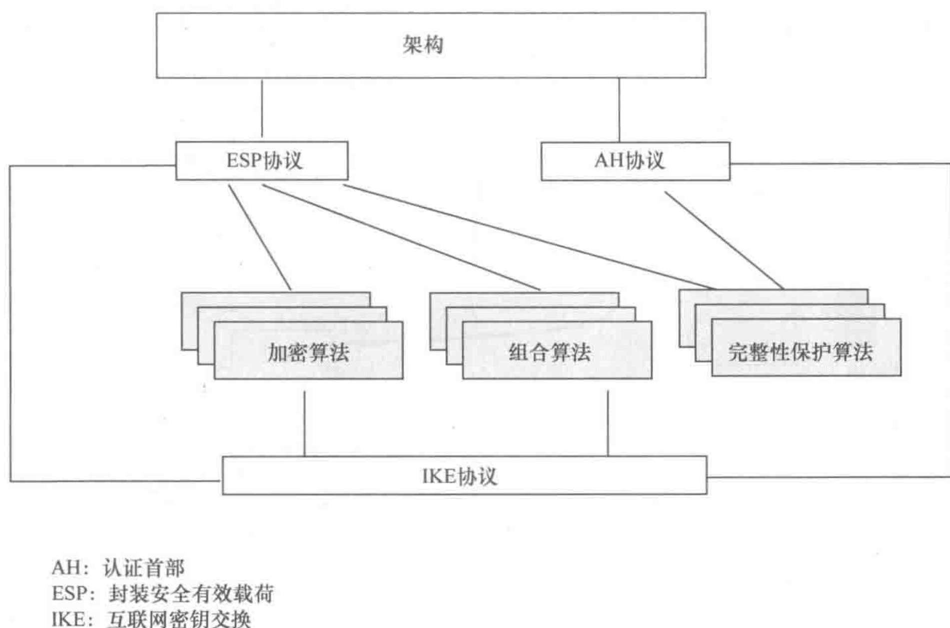


图 4.38　IPsec 规范之间的关系（引自 RFC 6071）

密码学（参见本章参考文献［22］，不仅可以了解相关算法，而且还可以了解相关的实现代码）为这些服务提供了基础。为此，IPsec 支持一系列的加密算法，目的是让两个相关端点具有共同支

持的算法的可能性最大化[1]。它还允许在新算法出现时增加新的算法，从而替代被发现有缺陷的算法。例如，RFC 7321 中规定的最新的加密算法 IPsec 要求，考虑到数据加密标准（Data Encryption Standard，DES）的弱点，强制要求支持高级加密标准（Advanced Encryption Standard，AES），并考虑到消息摘要5（Message Digest 5，MD5）的弱点，强制要求支持基于安全哈希算法 -1（Secure Hash Algorithm -1，SHA -1）的用于消息认证的密钥哈希（Keyed - Hashing for Message Authentication，HMAC）。这种开放、全面且最先进的加密技术在今天看来似乎是自然而然的事情，但实际上这需要很多年才能实现，这在一定程度上是因为不同国家的政府限制。密码学一直被认为是军事资源的一部分，它的出口受到政府的管制。商业和民用用户，如果没有被禁止使用加密技术，那么他们仅限于使用较弱且存在不足的算法。一些政府甚至要求密钥托管（即向当局披露密钥）。鉴于互联网的全球性质，这种情况无助于互联网安全，并且促使 IETF 发布信息 RFC 以表达他们的担忧。该 RFC 被分配的编号特意定为 1984[2]。

IPsec 的核心是安全关联概念。安全关联是一种加密保护的单工端到端会话。因此，双向会话至少需要两个安全关联。建立一个安全关联涉及产生加密密钥，选择算法，并选择不同的参数。在单播中，安全关联可通过被称为安全参数索引（Security Parameters Index，SPI）的参数进行识别。

IPsec 由两部分组成：安全关联管理和分组转换。安全关联管理部分处理 IPsec 端点验证，然后建立和维护安全关联。这里的一个基本的步骤是，安全服务的协商和相关安全参数的协议。整个过程在 IPsec 会话开始时执行，然后一段时间后，进行现场检查。相应的协议是复杂的，它的定义已经经历了多次迭代。它当前的版本，互联网密钥交换协议版本 2（Internet Key Exchange Protocol version 2，IKEv2）规定在 RFC 7296 中。为了生动地说明 IKE 的工作原理（以及它应该如何工作），本章参考文献[21] 提供了详细的描述。

分组转换部分处理每个分组事先约定的加密算法的实际应用。这个任务有两个不同的协议（遗憾的是，这反映出了人们都想推出自己的标准，而忽略了实际应用的必要性）。一个协议，被称为认证首部（Authentication Header，AH），在 RFC 4302 中定义。AH 提供了无连接的完整性保护、数据源认证和防重放服务。该协议不提供机密性保护。

相比之下，第二个协议，RFC 4303 中定义的封装安全有效载荷（Encapsulating Security Payload，ESP）协议，确实提供了机密性保护以及其他所有的安全服务。因此，ESP 的应用比 AH 要广泛得多。ESP 是计算密集型的，因为它需要执行所有的这些加密工作。可以使用硬件加速，由于操作系统最了解如何处理这种硬件，因此在主要的操作系统中，ESP 已经被实现成为这类操作系统内核的一部分。

通过 ESP，可以在 3 种情况下进行安全通信，如图 4.39 所示。

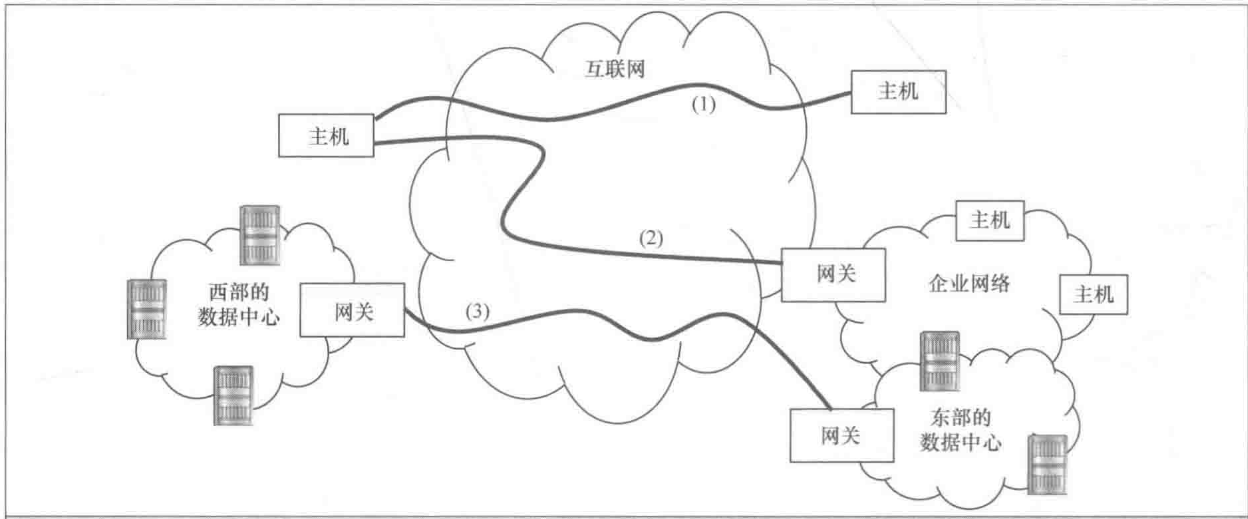


图 4.39　IPsec 应用情况

[1] NULL 加密算法，顾名思义，它的加密方式就是为了开发人员的方便起见而被包含在内的。它甚至有自己的 RFC 2410，感兴趣的读者可以阅读一下（引语："尽管有传言说，国家安全局禁止发布这种算法，但是没有证据表明他们这样做了。相反，最近的考古证据证明，NULL 算法创建于罗马时代，被视为凯撒密码的一种输出形式"）。

[2] 作为对 RFC 1984 的回应，IETF 后来发布了 RFC 2804，声明了对窃听标准的立场。

第一种情况是主机到主机。两个相互认证的主机通过互联网使用它们的公网 IP 地址彼此建立会话。外部人员无法窃听或改变它们之间的通信分组流，就好像使用专用通信线路一样。

第二种情况是主机到网关。远程主机通过互联网连接到企业网络的安全网关。网关可以访问网络中的所有其他机器，因此隧道一旦建立，远程主机也可以与它们进行通信。这种情况是典型的企业 VPN 访问（暂时回顾一下 VPN 方面的内容，正是这种机制，实现了安全的 IP - over - IP VPN。专用电话线路在这里被 IPsec 隧道所替代）。与第一种情况一样，其中的业务流量受到保护，但是与第一种情况不同的是，该主机可以拥有另一个 IP 地址，该地址仅在企业内部发挥重要作用。在这种情况下，网关也有两个 IP 地址，一个用于互联网其余部分的主机，一个用于企业。在下一章讨论了 NAT 之后，这一点的意义将会变得很清楚。

第三种也是最后一种情况是网关到网关。两个远程企业园区网络在互联网上通过它们各自的安全网关被结合成一个整体。这种情况与云计算特别相关，因为两个数据中心可以用完全相同的方式进行互连。在本质上，这种情况类似于第二种情况，唯一的区别是，隧道承载聚合而不是特定主机的业务流量。

IPsec 支持两种操作模式：传输和隧道。在传输模式下，每个数据分组只有一个 IP 首部，并且 IPsec 首部（AH 或 ESP）就在其后插入。为了指示 IPsec 首部的存在，IP 首部的协议字段采用 AH 或 ESP 的协议号。图 4.40 描述了 IPv4 传输模式下 IPsec 的分组结构。对于 AH 分组，完整性检查仅在 IP 首部中的不变字段上进行，例如版本、协议和源地址字段[㊀]。完整性检查值作为认证首部的一部分，该首部中还包含 SPI 和序列号。SPI 用作安全关联数据库条目的索引，该条目包含用于验证的 AH 分组的参数（例如，消息认证算法和密钥）。每个分组唯一的序列号的目的是用于重放保护。需要注意的是，IPsec 首部不包含有关正在使用的模式方面的信息，该信息存储在安全关联数据库中。

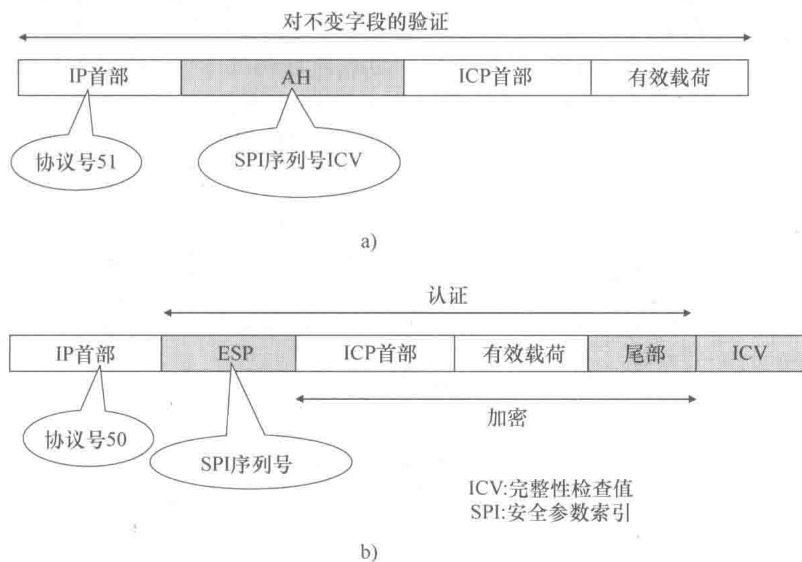


图 4.40　IPv4 传输模式中的 IPsec

对于 ESP 分组，完整性检查覆盖了 IP 首部之外的所有内容。此外，完整性检查值不作为 ESP 首部的一部分。它在一个单独的字段中提供。

在隧道模式下，有两个 IP 首部：携带最终 IP 源地址和目的地址的“内部” IP 首部，以及携带 IPsec 端点地址的“外部” IP 首部。图 4.41 描述了相应的分组结构 [图 4.41a 有一定的说明意义，但不完全正确。除了所示的不变字段之外，还有其他不变字段，例如源 IP 地址]。

㊀ 这里有一个严重的复杂问题，它是由 NAT 引起的，因此，稍后在讨论这些问题的时候再对其加以讨论。

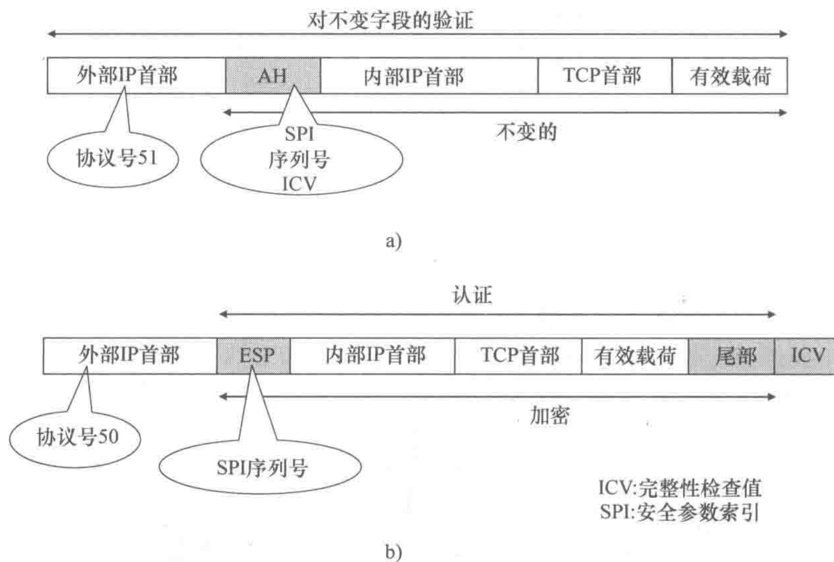


图 4.41 IPv4 隧道模式中的 IPsec

考虑到 ESP 可以做所有 AH 能做的事情，但 AH 却不能做所有 ESP 能做的事情，因此人们想知道 AH 的价值何在。一种观点是，AH 通过保护 IP 首部本身提供了略微更多的保护。另一种观点是，AH 与防火墙的配合更好（稍后再讨论这方面的内容）。还有一些其他的观点，但最终最重要的是：使用 IPsec 的主要原因是机密性保护，而这不是 AH 可以提供的服务。就 IPsec 方案而言，传输模式适用于第一种情况，而隧道模式则适用于其他情况。

到目前为止，已经讨论了 IPv4 下 IPsec 的操作。只能说 IPv6 下 IPsec 的操作与 IPv4 下并没有太大的区别，当然，除了首部。鉴于本书篇幅有限，不能深入研究 IPv6 的复杂性内容。再一次强调，本章参考文献［21］在这方面提供了翔实的内容，阐明了 IETF 策略的各个方面。

参考文献

[1] Tanenbaum, A. S. and Van Steen, M. (2006) Distributed Systems: Principles and Paradigms, 2nd edn. Prentice Hall, Englewood Cliffs, NJ.

[2] Birman, K. P. (2012) Guide to Reliable Distributed Systems: Building High – Assurance Applications and Cloud – Hosted Services. Springer – Verlag, London.

[3] Faynberg, I., Lu, H. – L., and Gabuzda, L. (2000) Converged Networks and Services: Internetworking IP and the PSTN. John Wiley & Sons, New York.

[4] International Organization for Standardization (1994) International Standard ISO/IEC 7498 – 1: Information Technology—Open Systems Interconnection—Basic Reference Model: The Basic Model. International Organization for Standardization/International Electrotechnical Commission (ISO/IEC), Geneva. (Also published by the International Telecommunication Union—Telecommunication Standardization Sector (ITU – T) as ITU – T Recommendation X. 200 (1994 E).

[5] Tanenbaum, A. S. and Wetherall, D. J. (2011) Computer Networks, 5th edn. Prentice Hall, Boston, MA.

[6] ITU – T (1996) ITU – T Recommendation X. 25 (formerly CCITT), Interface between Data Terminal Equipment (DTE) and Data Circuit – terminating Equipment (DCE) for terminals operating in the packet mode and connected to public data networks by dedicated circuit. International Telecommunication Union, Geneva.

[7] Halsey, J. R., Hardy, L. E., and Powning, L. F. (1979) Public data networks: Their evolution, interfaces, and status. IBM Systems Journal, 18 (2), 223 – 243.

[8] Metcalfe, R. M. and Boggs, D. (1976) Ethernet: Distributed packet switching for local computer networks. Communications of the ACM, 19 (7), 395 – 405.

[9] Cerf, V. and Kahn, R. (1974) A protocol for packet network intercommunication. IEEE Transactions on Communications, 4 (5), 637 – 648.



[10] Information Sciences Institute, University of South California (1981) DoD Standard Internet Protocol. (Published by the IETF as RFC 791), Marina Dely Rey.

[11] Huston, G. (1999) Interconnection, peering and settlements—Part II. The Internet Protocol Technical Journal, 2 (2), 2 - 23.

[12] Turner, J. S. (1986) Design of an integrated services packet network. IEEE Journal on Selected Areas in Communications, SAC - 4 (A), 1373 - 1380.

[13] Clark, D. D., Shenker, S. S., and Zhang, L. (1992) Supporting real - time applications in an integrated services packet network: Architecture and mechanism. SIGCOMM '92 Conference Proceedings on Communications Architectures & Protocols, pp. 14 - 26.

[14] Stiliadis, D. and Varma, A. (1998) Latency - rate servers: A general model for analysis of traffic scheduling algorithms. IEEE/ACM Transactions on Networking (TON), 6 (5), 611 - 662.

[15] Rekhter, Y., Davie, B., Rosen, E., et al. (1997) Tag switching architecture overview. Proceedings of the IEEE, 85 (12), 1973 - 1983.

[16] Lu, H. - L. and Faynberg, I. (2003) An architectural framework for support of quality of service in packet networks. Communications Magazine, IEEE, 41 (6), 98 - 105.

[17] Farrel, A. and Bryskin, I. (2006) GMPLS: Architecture and Applications. The Morgan Kaufmann Series in Networking. Elsevier, San Francisco, CA.

[18] Faynberg, I., Gabuzda, L. R., Kaplan, M. P., and Shah, N. (1996) The Intelligent Network Standards: Their Applications to Services. McGraw - Hill, New York.

[19] Lakshman, T., Nandagopal, T., Sabnani, K., and Woo, T. (2004) The SoftRouter Architecture. ACM SIGCOM HotNets, San Diego, CA. http://conferences.sigcomm.org/hotnets/2004/HotNets - III%20Proceedings/lakshman.pdf.

[20] McKeown, N., Anderson, T., Balakrishnan, H., et al. (2008) OpenFlow: Enabling innovation in campus networks. ACM SIGCOMM Computer Communication Review, 38 (2), 69 - 74.

[21] Kaufman, C., Perlman, R., and Speciner, M. (2002) Network Security: Private Communications in a Public World. Prentice Hall PTR, Upper Saddle River, NJ.

[22] Schneier, B. (1995) Applied Cryptography: Protocols, Algorithms, and Source Code in C. John & Wiley Sons, New York.

第5章 网络设备

这里描述的所有设备都是现代数据中心的组成部分。它们既支持网络边界的构建，又支持应用部署。以物理和虚拟的形式，它们还支持云计算，因为网络是其基本组件。

5.1 节描述的第1种关键网络设备是域名系统（Domain Name System，DNS）服务器。为了访问互联网上的任何资源（网页、邮箱、网络电话），人们最终需要指定资源的 IP 地址。然而，应用不应该知道资源的 IP 地址（且很少这么做）。相反，它使用资源定位符——根据应用层命名方案指定的字符串。然后，定位器将被 DNS 实时转换为相关的 IP 地址。这种解决方案的优势是除了支持人们易记的名称之外，它还支持资源移动性，且可用于支持弹性以及用于负载均衡。也就是说，在多个执行同一操作的服务器之间分配工作量。但是，转换服务并非 DNS 的唯一优点，它还可用于服务发现，这使得 DNS 基础设施对于网络和云计算尤为重要。

本章中描述的第2种设备称为防火墙。防火墙是保障安全的主要手段，特别是在网络隔离和保护方面。

接下来讨论的第3种设备是网络地址转换（Network Address Translation，NAT）设备。该设备对于组网来说必不可少，但业界仍存有争议，主要用于解决快速耗尽的 IPv4 地址空间问题。虽然网络地址转换几乎完全在防火墙中实现，但其功能却不尽相同。

在本章的最后，简要回顾了负载均衡器设备，它在很大程度上支持云的弹性。由于本书篇幅有限，因而仅讨论了由 DNS 服务器执行的不同风格的负载均衡。

5.1 域名系统

开发域名系统（DNS）的主要动力来自于电子邮件，或者是其特定的实现。虽然自20世纪60年代中期以来电子邮件就一直作为一种服务存在，但需要强调的是，Ray Tomlinson 在为 TENEX 操作系统开发电子邮件包时，定义了一种资源定位器，形式为 <user>@<host>，其中“@”字符用于将 ASCII（American Standard Code for Information Interchange，美国信息交换标准代码）编码的用户名和主机名分隔开来。这种形式的寻址于1972年在阿帕网（ARPANET）中变成标准，当时电子邮件支持被并入到文件传输协议（File Transfer Protocol，FTP）中，并体现在 RFC 385 中。

当然，为了使电子邮件正常发挥作用，必须将主机名转换成正确的 IP 地址。每台主机都保存了一份转换文件的副本，这在当时（1971年）自然不是什么大问题，因为阿帕网（ARPANET）中只有23台主机。但是，两年后（1973年），Peter Deutsch 在 RFC 606 中写道：“既然我们拥有了主机名的正式列表，似乎是时候该结束这种荒谬状况了，即为了使用自己的操作系统或用户程序，网络上每个站点都必须维护一个不同的（通常是过时的）主机列表。”

事实上，这种“荒谬状况”已经结束了。在用于响应 RFC 606[⊖]所写的几份 RFC（Request For Comments，请求注解）之后，发布于1974年3月7日篇幅只有1页的 RFC 625 证实，斯坦福研究所（现在成为 SRI 公司）将保存主机转换文件的主副本（称为 HOST. txt），它对其他主机来说是可用的。RFC 625 坚持认为：①文件以 ASCII 文本形式保存；②使用 FTP 来访问文件。

⊖ 当时，RFC 系列看起来不仅提供信息或提出要求，而且还可为开发社区充当讨论媒介。

7年后，这一解决方案也开始看起来非常荒唐，但有一个非常不同的原因。罪魁祸首是电子邮件寻址系统，现在已经实现了标准化。在该系统中，发件人必须将整个中继路径提供给接收机㊀。这一过程不仅麻烦，而且对于任何人来说，当给定路径失败时，要找到正确的路径在技术上是不可能的。

DNS的历史可追溯到1982年1月11日，当时有22位互联网工程师参加了Jon Postel㊁召集的一次会议，“主要讨论ARPA（Advanced Research Project Agency，高级研究项目局）互联网文本邮件系统中的一些具体问题”。上述引用来自RFC 805，该文档详细记录了当时的讨论和决定。文档提到的第一个具体问题是电子邮件的中继，RFC列出了若干个提案，指出“此次讨论中出现的一个有趣的想法是邮箱标识符的user@ host模型原则上应当可以被unique - id@ location - id模型所替换，其中unique - id是该邮箱的全局唯一标识符（Globally Unique Identifier，GUID），它与位置无关，且location - id是在何处发现邮箱的建议。”这一想法没有得到追捧，因为“... ... user@ host模型为大家广为接受已经是不争的事实……所以‘用户’域的诸多不同阐述已经被使用。”㊂

然而，大家一致认为：“当前的user@ host邮箱标识符应当扩展到user@ host. domain，其中domain可以是域的层次结构。特别地，‘主机’字段将成为‘位置’字段，且结构将从最具体到最通用的方式读取（从左到右）。”这是专用数据库（域名服务器）的职责，用于存储与域相关的信息，并在查询时提供信息。在这一愿景中，最显著的部分是（且在我们看来，目前仍然是）将服务与提供该服务的主机分离开来的概念。最终，25年后，这一愿景促使了万维网（World Wide Web，WWW）的实现，接着是IP电话、IPTV（IP Television，IP电视）以及最终云的实现。仅将电子邮件作为坚实的参考点，设想一种支持众多抽象服务的系统是符合“弗拉基米尔·纳博科夫（Vladimir Nabokov）”小说《礼物》[1]中给出的天才定义：“天才是能够凭空想象出雪的非洲人。”RFC 882宣称：“我们应该能够使用名称来获取主机地址、邮箱数据和其他迄今为止尚未确定的信息。”

关于所设想的查询，确定了3种独立服务，但协议是只有这些服务中的一种服务（将主机名位置ID转换为相应IP地址的服务）至关重要。大家还认识到，域名服务器可以返回其他信息（如主机上使用邮件过程中的信息），虽然这些信息也可以通过其他方式来获得。

关键的架构决策是针对名称查询的集中式对分布式实现问题做出的：“事实是为每个域分配独立服务器具有管理和维护方面的优势，但是中央服务器可能是有用的第一步。人们还认识到，应该将每种不同的数据库复制若干次，且可以从不同的服务器获得，用于鲁棒性和可靠性服务。”在该文档（RFC882）中，递归域名服务器首次以实例情境的形式出现：

“假设新邮箱规范的格式为USER@ HOST. ORG. DOMAIN，如Postel@ F. ISI. IN。向该地址发送邮件的源主机首先查询域IN的域名服务器（给出整体位置F. ISI. IN）。查询的结果要么是目标主机的最终地址（F. ISI），要么是ISI（Information Sciences Institute，美国信息科学研究所）域名服务器的地址，要么是ISI转发器的地址。在第1和第3两种情况下，源主机将邮件发送到返回的地址。在第2种情况中，源主机查询ISI域名服务器并...（递归调用本段）。”

由于1982年1月1日Jon Postel召集的会议对系统提出了一些关键需求，因而RFC 882和RFC 883分别对系统的设计原则和实施考虑进行了规范。这些RFC文档均是由Paul Mockapetris起草的，他负责设计DNS的任务。这两份文档㊃均于1983年11月发布，同时Jon Postel发布了项目管理文档——RFC 881，计划推出阿帕网（ARPANET）中的DNS，并在1年后（1984年12月）停止维护HOST. TXT文件。

DNS软件第1版——伯克利互联网域名（BIND），是由加州大学伯克利分校发布的。BIND的后期版本由数字设备公司（Digital Equipment Corporation，DEC）编写。最终，BIND的发展并入到互联网系

㊀ 1981年，Jon Postel在RFC 788中发布了简单邮件传输协议（Simple Mail Transfer Protocol，SMTP）第1版。通过明确指定中继路径来实现电子邮件的中继。（RFC给出了“TO:”字段的实例为：<@ A，@ B，C@ D。）

㊁ Postel博士和Mockapetris博士是领导域名系统（DNS）开发的两个人。因此，总体思想和概念显然是由Postel博士提出来的，而Mockapetris博士负责设计和后续项目管理。

㊂ 实际上，这一想法是将地址变成两个标识符的组合：统一资源名称（Uniform Resource Name，URN）——邮箱的唯一电子邮件标识符以及统一资源定位符（Uniform Resource Locator，URL）。很快就会看到，这些标识符后来是被独立开发的。

㊃ 这些RFC文档已经被RFC 1034和RFC 1035取代。

统协会（Internet Systems Consortium，ISC）中。目前，BIND 是一个开源项目。

在随后几年中，这些标准及其实现方案不断演进，但这些变化是渐进的，除了安全性方面的一个主要变化。事实上，安全性起初既不是一项要求，又不是一个问题（而可扩展性是）。因此，安全性是一个附加项。遗憾的是，可以将安全性指定为利用严重漏洞后的一种反应。2005 年，IETF（Internet Engineering Task Force，互联网工程任务组）发布了一种名为 DNSSEC（Domain Name System Security Extensions，域名系统安全扩展）的安全解决方案（稍后将会对其进行详细讨论），尽管部署方式存在严重问题。DNSSEC 也只是部分解决方案。正如将要看到的那样，防火墙可以应对诸多威胁，但仍然依赖于（而且还将依赖于）网络提供商及其客户对安全实践的一致实现方案。为此，云可以一致地实现和执行安全策略。

本节的剩余部分将介绍域名系统的架构和操作，然后再重点介绍顶级域名分类和域名的国际化问题。最后一小节讨论安全问题和域名系统安全扩展（DNSSEC）。

5.1.1 架构和协议 ★★★

域名系统（DNS）架构的组件如图 5.1 所示。最接近终端用户的组件是用于应用进程查询的解析器。解析器有望成为（且通常是）操作系统的一部分，因而在程序进程和解析器之间不存在协议，且解析器是针对两者之间的接口定义的。这是一个应用程序接口问题。

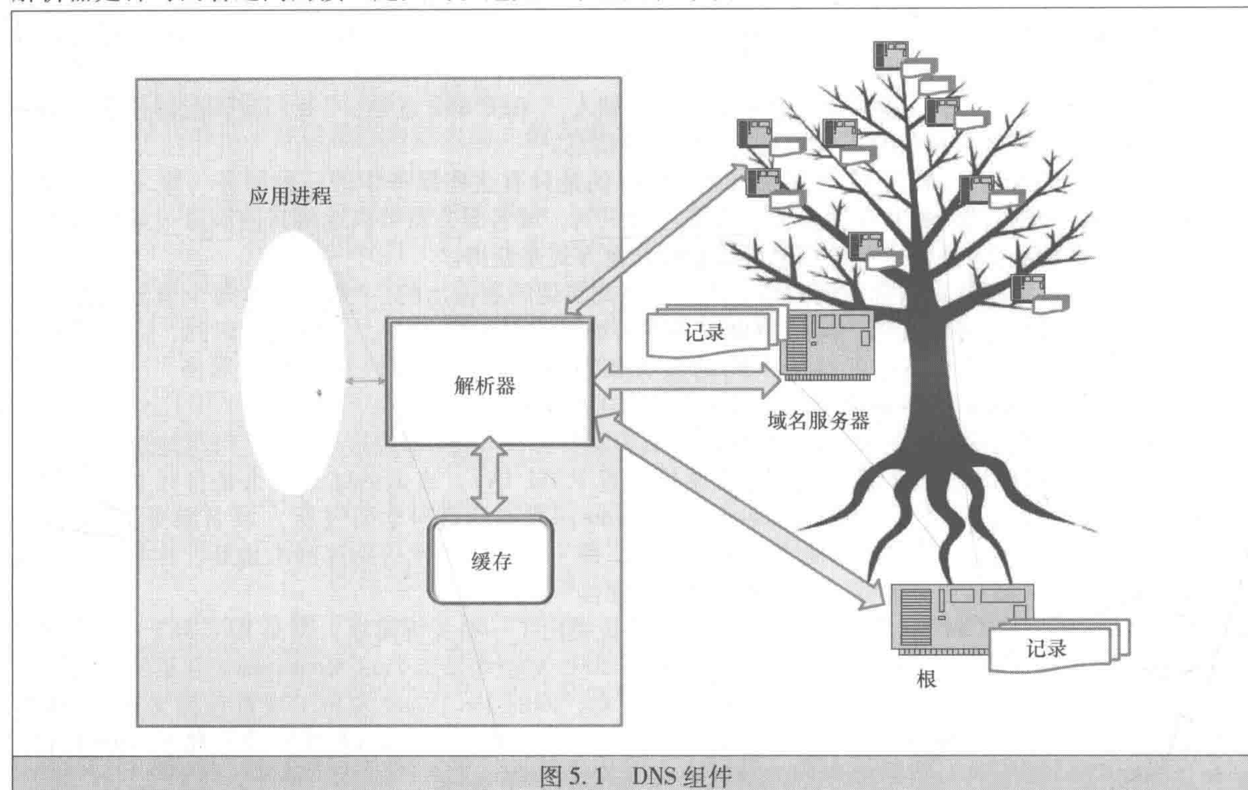


图 5.1 DNS 组件

反过来，解析器通过查询一台或多台域名服务器来获取进程所需的信息。将响应存储在本地缓存中，因为期望特定域中存在多个资源引用是非常自然的。缓存将信息保持一段时间，但不超过它所存储的记录生存时间（Time To Live，TTL）参数值。当查询到达解析器时，它首先检查高速缓存中是否有记录。如果没有，则解析器开始使用 DNS 协议［顺便一提，通过 UDP（User Datagram Protocol，用户数据报协议）操作，使用标准端口 50[㊀]］来查询域名服务器。

域名服务器实现了域名空间，且它们包含与域名相关的资源记录。DNS 是针对命名空间随时间而出现的指数级增长。因此，域名空间的实现方案采用了树结构。正如 RFC 1034 所述，“从概念上讲，域名空间树的每个节点和叶子均命名了一组信息，且查询操作试图从特定集合中提取特定类型的

㊀ 对于长查询来说，通常使用 TCP（Transfer Control Protocol，传输控制协议），但这是一个例外。



信息。”

图5.2进一步解释了这一模型。域名服务器负责域空间子集（称为区域）进行处理，且将该域名服务器称为该区域的权威（或授权域名服务器）。除了提供与区域相关的信息之外，域名服务器还提供其他域名服务器的地址（在其区域之外）。

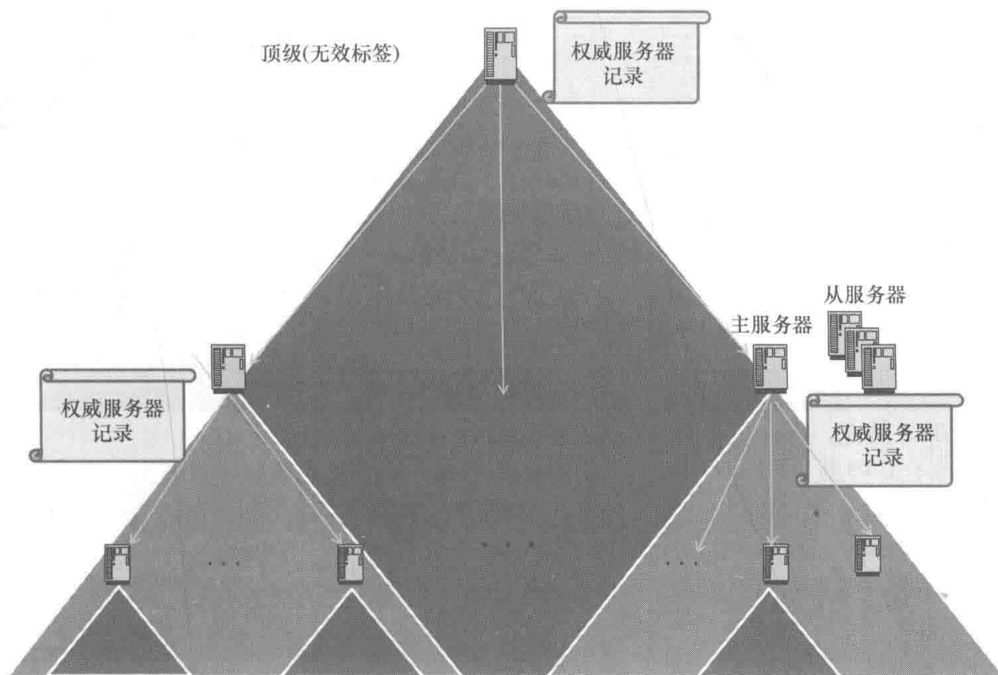


图5.2 域名空间树

区域管理员可以创建其他区域，从而在树中创建另一分支。根据RFC 1034文档，“数据库在特定机构想要接管子树控制权的点进行分区。一旦机构控制了自己的区域，则它可以单方面更改区域中的数据，增加连接到区域的新树部分，删除现有节点，或者为其区域下的新分区授权。”

为了确保可靠性，通过在区域中保存若干个相同从域名服务器来引入冗余，这些都是由主服务器自动进行更新的。

域名的语法（它是一个包含字母和数字的字符串）严格对应于域名空间树，因为每种资源的定位符都会从树根中得出完整路径。树的每个节点都拥有一个标签（长度最多为63字节）。需要注意的是，根标签的长度为0——它是一个空字符串。域名是从右到左拼成（根据源于早期电子邮件寻址的长期公认的惯例）的一系列标签。因此，最右边的标签总是对应于顶级域。

标签是不分大小写的[㊀]。根据定义，主机名是至少拥有一个与其关联的IP地址的域名。并非每个域名都是主机名（如.com或.edu都不是主机名）。通过域名国际化，域名的表达方式已经发展到适应多种语言及其各自的字母表。

迄今为止，仅在解析器的情境中讨论了DNS的客户端，它不断查询服务器[㊁]，直至找到地址。但是平滑网络流量的需求（特别是当简短查询存在的情况下）要求引入可以工作于“递归模式”的域名服务器（即本身可充当客户端，在返回答案之前查询多台服务器）。递归域名服务器（见图5.3）也缓存了答案。遗憾的是，缓存保存和开放递归服务配置已成为安全攻击的引擎，将在本章后面进行介绍。

域名服务器的默认（和强制实现）模式是非递归的。在这一模式下，它仅使用本地信息来回答查询。它通过提供答案、错误或推荐另一台服务器来做出响应。要使用递归模式，客户端和服务器都必须达成共识。

㊀ 请参阅RFC 4343（txt）中与转义序列和其他棘手问题相关的说明。
㊁ 如前所述，该过程（本质上是迭代过程）早期在DNS定义中被称为递归，且名称已经不再使用。

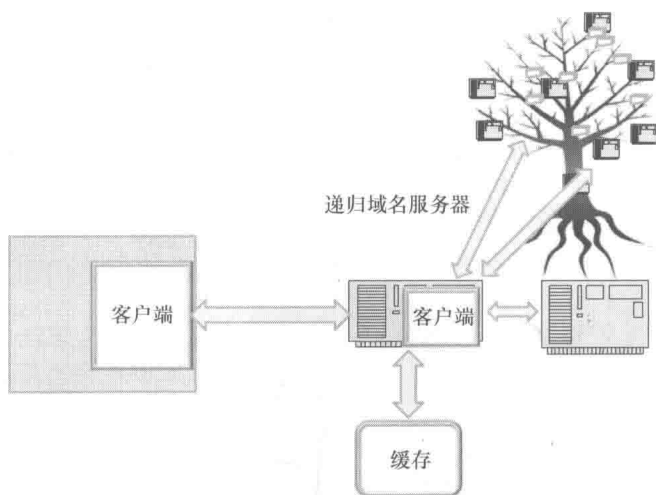


图 5.3　递归域名服务器

为了说明 DNS 操作，将介绍 DNS 协议的基础知识。

图 5.4 给出了 DNS 查询和 DNS 响应的通用格式。16bit 边界对齐是协议数据单元的主要组成部分。

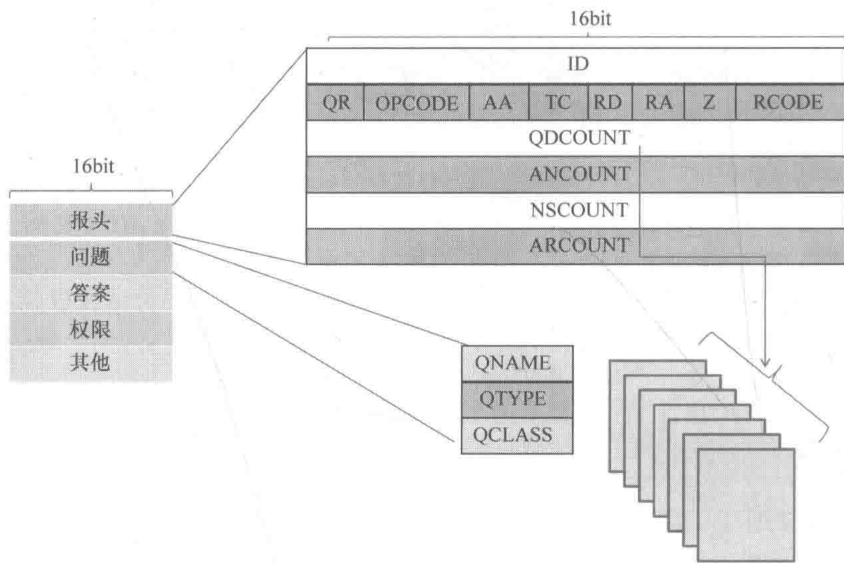


图 5.4　DNS 查询/响应格式（引自 RFC 1035）

最相关的部分是报头，其中包含：

1）单元标识符（Identification，ID），以便将回复与未完成的查询相关联；

2）标签位（QR），它表示该单元是查询还是响应；

3）OPCODE，用于指定查询[㊀]；

4）标志（AA），仅在表明响应是否具有经过授权的响应中才有意义；

5）截断标志（如果消息因超过域允许的长度而已被截断，则对该字段进行设置）；

6）两个标志——期望递归（Recursion Desired，RD）和可用递归（Recursion Available，RA），其中前者仅用于指示域名服务器递归执行查询的查询，而后者用于表示服务器是否真正支持递归查询的

㊀ 例如，DNS 最初设计用于支持反向查询，但该功能已被弃用。

响应中——Z 位预留供将来使用，且预期值为 0；

7）RCODE，用于指定响应代码；

8）QDCOUNT、ANCOUNT、NSCOUNT 和 ARCOUNT，分别指定后面各部分（问题、答案、权限和其他）中各项代表的整数。

问题部分是包含 QNAME（域名）、QTYPE（将在讨论资源记录时提供更多内容）和 QCLASS 的记录数组（大小为 QDCOUNT），该数组实际上包含一个 IN 值（对于互联网来说）[⊖]。

答案、权限和其他部分是相同的，因为每个字段都包含一组资源记录（Resource Record，RR）。

资源记录（RR）的结构如图 5.5 所示，其中的绝大部分是不解自明的。



图 5.5 资源记录（RR）的结构（引自 RFC 1035）

记录 TYPE 是一个域，它已经被赋予新值。一个实例是 A（针对 IPv4 地址），后来由 AAAA 来补充（针对 IPv6 地址）。其他实例是起始授权机构（Start Of Authority，SOA）、CERT（针对加密证书）和 SERV（针对服务）。SERV 由 RFC 2782 指定，支持我们明确执行指定服务的服务器位置。此功能对于电子邮件和 IP 电话非常有用，其中资源定位器（本节稍后将进行解释）可用于发现特定的域名提供服务。

使用字符串可能会变得更加复杂（正如所有编译器作者都知道的）。图 5.6 中描述的一个问题是循环依赖关系，当域名服务器返回指向待解析原始域的名称引用时，会出现循环依赖问题。为此，RFC 1035 假设在这种情况下，在引用中指定了胶水记录（即实际地址）。

生存时间（TTL）参数是一个有趣的实例，必须对早期（1983 年）规范进行修改。人们发现，16 位整数太小而无法指定合适的时间值（单位为 s），因而将该字段加长到 32 位。

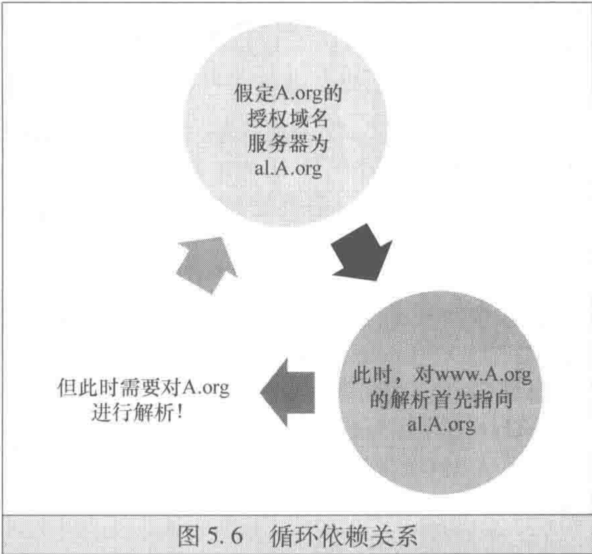


图 5.6 循环依赖关系

⊖ IETF 不鼓励在 RFC 2826 中使用备用根域名服务器，且饱受争议的 AlterNIC 和 eDNS 根服务器已被关闭。人们仍然在努力构建备用根服务器，其中的一台备用根服务器用于“位”比特币操作。

5.1.2 DNS 操作 ★★★

迄今为止，根据所描述的内容，应该清楚的是运行顶级域名服务器是一项艰巨的技术任务（因为需要将所有数据库复制到网络外界，且同时确保安全）。但除了技术复杂性之外，域名可以进行买卖的事实自然使事情变得更为复杂。

DNS 被采纳后，Jon Postel 成立了互联网号码分配机构（Internet Assigned Numbers Authority，IANA）。除了其他任务之外，IANA 的一项重要任务是管理根域。1998 年，IANA 成为新成立的互联网名称与号码分配机构（Internet Corporation for Assigned Names and Numbers，ICANN）的一个部门。根据美国政府的某项商业合同，除了执行其他服务之外，条款规定：该部门还需要“①协调技术协议参数的分配，包括管理地址和路由参数区（Address and Routing Parameter Area，ARPA）顶级域；②管理与互联网 DNS 根域管理相关的某些责任，如通用（gTLD）和国家代码（ccTLD）顶级域名；③互联网编号资源的分配……”

顶级域名（Top Level Domain，TLD）对应于 DNS 根域。将所有 TLD 的相关信息复制到 13 台根服务器上（更确切地说，是根系统，稍后将加以解释）中，由字母表的相应字母进行命名——从 A 到 M。每台服务器都有一个名称，从该字母开始，并加上后缀“. root - servers. net”。

认为服务器是单一主机是一种常见的错误（尽管很久以前就出现了这种错误）。除了一个例外㊀，这些主机在世界各地不同服务器之间进行复制，这些服务器的总数目前接近 400 台。因此，“服务器”这个词不够准确，它实际上指的是系统。（仅 L 系统就拥有覆盖全球的 146 台服务器！）事实上，大多数根系统作为网络运行——它们具有自治域（Autonomous System，AS）号码，并输入对等协议。通常，任意播 IP 寻址用于查找最接近查询发布者的计算机。

在顶级域中，不存在主系统或主服务器㊁。所有根系统（更确切地说，每个根系统中的实际主机）在所有现存顶级域授权的域名服务器的地址记录方面具有相同的知识。管理机构——威瑞信公司（扮演承包商的角色）使用它所维护的同一组文件对所有主机进行更新。DNS 根服务器在美国商务部（Department of Commerce，DoC）的下属机构——国家电信和信息管理局（National Telecommunications and Information Administration，NTIA）的授权下运行。

5.1.3 顶级域名标签 ★★★

edu 是顶级域名（TLD）的一个实例，它预留给可信任的高等院校使用。mil 和 gov 是具有严控注册权限的顶级域名的其他实例。这些域名分别预留给美国军方和美国政府机构。一些诸如 com 或 biz 等域名（主要用于商业领域），最初预留用于某一目的（在实例中，预留给商业公司），但最终对所有人开放。类似的情况还有 net 域名，它最初是为了指定网络而创建的，但最终被用于各种目的。

在 1998 年左右，控制顶级域名的发展不太顺利——随着电子商务的发展，不同团体（包括一群认为互联网属于自己的 IETF 工程师和互联网爱好者）声称互联网资源属于自己。这里既没有篇幅也没有时间来描述这段历史。需要注意的是，ICANN 是为管理顶级域名而产生的。（现在，大家或许会对前面引用的关于 ICANN 使命陈述的摘录有更加深入的理解。）

ICANN 对几组顶级域名进行了区分。通用顶级域名（Generic Top Level Domain，gTLD）包括最常用的标签，如 com、net、edu 等。国家/地区代码顶级域名（country code Top Level Domain，ccTLD）基于双字符 ISO（International Organization for Standardization，国际标准化组织）国家/地区代码。基础设施顶级域名（使用名称 arpa，读者可能还记得 ICANN 的使命陈述）主要用于反向查询 IP 地址。

ccTLD 还存在于各自的国际化版本中（将在后面进行讨论），以及为测试国际化而创建的特殊测试域。最后，RFC 2606 定义了仅用于测试、自引用和引用的域作为实例㊂：“…预留了 4 种域名，将其描述如下：

1）建议将 . test 用于测试当前或新的 DNS 相关代码；

㊀ B. root - servers. net 拥有单一位置。

㊁ 在 DNS 的早期，A 服务器（那时它仅仅是服务器）是主服务器。

㊂ 这样做是为了避免版权和商标的违规行为，并防止人们在实例中使用“真”域名时引起其他潜在法律问题。

2）建议将.example用于文档或实例中；

3）invalid开发用于域名的在线构建，这些域名注定无效，且显而易见是无效的；

4）按照传统做法，通常在主机DNS实现方案中静态定义.localhost，因为它拥有指向环回IP地址的A记录，并预留用于此用途。”

如前所述，一些gTLD（如.com或.net）最终允许任何人和任何事的注册，因而仅根据标签来判断，不可能从域下注册的实体推断出更多信息。相对于赞助顶级域名（如.edu、.mil或.int），将这种通用顶级域名称为非赞助顶级域名。赞助顶级域名预留给由基于条约成立的国际组织，它们拥有与此类域相关的特定社区或兴趣组。对于赞助通用顶级域名（gTLD）来说，它是一种定义了能够确定注册资格并执行此类策略的社区。

要真正理解ccTLD并非一件容易的事情。在特定国家名称下注册需要做些什么？事实证明，没有简单的答案。一些国家要求注册公民提供身份证明（如代表阿尔巴尼亚的顶级域名.al）。其他国家（如代表爱沙尼亚的顶级域名.ee或代表德国的顶级域名.de）只要求当地行政联系人在国内有实体存在。仍然有一些国家（如厄立特里亚），根本不提供注册服务。各国都可以出售其顶级域名，这使得事情变得更加混乱。众所周知的实例是托克劳将其域名“.tk”出售给荷兰公司DOT.TK。据报道，戴伦·鲍利（Darren Pauli）在其网络文章中报道了主机垃圾邮件发送者之事。

域名历史上的重要发展里程碑之一就是它们的国际化，或者用户使用本国语言编写脚本的能力。这包括两方面：①本地脚本的编码，以便它可以通过浏览器显示在监视器上（如在中国使用汉语进行编码）；②脚本与实际DNS使用之间的互通。

自20世纪70年代以来，对终端和键盘上的脚本国际化的支持力度不断加大。目前，存在着各种各样的专有实现方案（针对各种制造商生产的计算机终端），但Unicode联盟针对此项用途开发了一种被称为Unicode的标准编码方案。至于与DNS的互通，作为IETF互联网草案的第一个提案显然是于1996年发布、由Dürst主笔的。

从那时起，域名国际化取得了很大的成就，主要标准——RFC 3490：应用中的域名国际化方案（Internationalizing Domain Names in Applications，IDNA）发布。首要想法是在不改变现有基础设施的情况下增加新功能。换句话说，只修改应用程序，而不修改DNS。为此，DNS仅使用ASCII编码（非Unicode编码）标签。

图5.7演示了组件，并说明了IDNA操作。

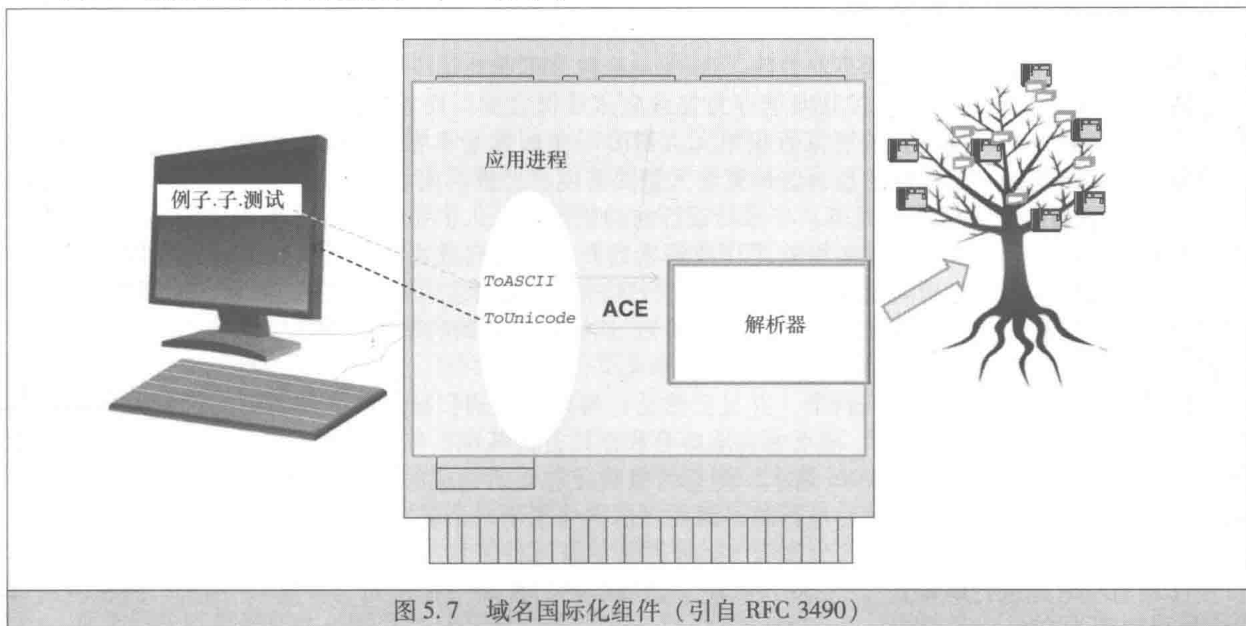


图5.7 域名国际化组件（引自RFC 3490）

当用户以他/她选择的任何语言或脚本键入域名时，应用程序（通常是浏览器）的工作是将Unicode标签转换为ASCII兼容编码（ASCII-Compatible Encoding，ACE）字符串，反之亦然。映射由Punicode指定，它是在2003年发布的RFC 3492中定义的，后来进行了更新。ToASCII算法将Unicode编码

的字符串转换为唯一的 ASCII 编码字符串，并将 4 字符前缀（“xn - -”）添加到结果中，因而所有 IDNA 标签都以这 4 个字符开头。相反，ToUnicode 算法剥离 ACE 前缀，并将其余部分转换为 Unicode 编码的字符串。

浏览器的工作是执行转换，使得从解析器开始的整个 DNS 仍然不受影响㊀。ToASCII 是在从用户接收到的字符串上执行的；而 ToUnicode 是在从解析器接收到的字符串上执行的。

国际化存在的主要问题之一（它与所有主要计算问题面临的代表性问题类似）是安全性，具体来说就是欺骗问题。即使使用明文 ASCII，某些字母和数字（如字母“l”和数字“1”）在某些字体中看起来也非常相似，这一点可能会被恶意攻击者所利用。

随着国际化步伐的加快，事情变得越来越严重。读者能够看出 www. paypal. com 和 www. paypal. com 之间的区别吗？当然不能。因为这两个字符串在纸张和计算机屏幕上看起来完全相同的，这正是问题所在——第 2 个域名实际上包含 1 个西里尔字符“a”，它代替了外形相同的拉丁字母字符。（字母是以 Unicode 格式输入本书的。）如果将对应于这些域名的字符串进行比较，则会发现这些字符串是截然不同的。这里将拼写相同但含义不同的词称为同形异义词。这一定义在计算中已经得到扩展，它表示看起来相同的字符串。《美国计算机学会通信》中的一篇文章[2]报道了一种现有钓鱼攻击，并警告其在国际化域名系统中的可能性。

如果所有浏览器均禁止使用属于同一标签中不同字母表的 Unicode 字符组合，则可能会阻止攻击，但这将非常难以实施。针对域名注册商，该解决方案通过拒绝注册含有此类混合字母的域名的方式，来禁止混合来自不同字母表字符的做法。这表明国际化域名的层次结构将在各自权威机构之下进行创建和配置。

创建 DNS 的独立顶级域名主要用于实现国际化。将其称为国际化国家/地区代码顶级域名（IDN ccTLD）。

ICANN 董事会于 2006 年 12 月批准设立了一个国际化顶级域名工作组。在不到 3 年的时间里，他们开始接受来自全球各国和地区代表的顶级国际化域名申请。

5.1.4 DNS 安全 

与早期互联网的发展一样，发明互联网的人首先开始使用互联网。所有工作的事物既令人惊异，又发展充分。没有人怀疑系统会被滥用（而且，在早期滥用它也不会产生明显的经济效益）。因此，所有应用所依赖的 DNS，在保护系统免受潜在攻击者威胁方面并未做任何特殊设计。这一也许过于乌托邦式的想法，使互联网社区永远是友善的。Jon Postel 在 RFC 760（1980 年发布）中提出其著名的鲁棒性原则：“通常来说，实现方案在其发送行为方面应该是保守的，而在接收行为中应该是自主的”。最后一个分句意味着如果接收到的协议数据单元（PDU）在形式上不够统一但可以解释，则应该接受协议数据单元而不是拒绝。遗憾的是，这种宽宏大量的态度已经被利用。

利用 DNS 漏洞的动机非常简单，与抢劫银行的动机类似：人们抢劫银行是因为银行里有钱。如果攻击者能以某种方式欺骗递归域名服务器（或解析器）接受指向攻击者网站（而不是银行网站）的错误记录，则他可以了解与访问过攻击者网站的银行客户有关的诸多信息（包括他或她的密码）。在许多客户身上重复同样的伎俩（这正是软件所擅长的），然后使用得到的密码来提钱，最终达到与抢劫银行相同的目的。

DNS 原始设计存在的主要问题是（并且仍然是）解析器或递归域名服务器得到的记录未被验证㊁。最终，只要某个响应与 QueryID、请求者的端口号和原始查询匹配，则认为该响应是有效的。如果某个进程（称为中间人）可以拦截 DNS 请求，则它可以响应进程所构成的任何记录。请求的发起者无法看到任何差异。创建中间人并不总是可行的，因而攻击进入了下一个复杂程度。

使用缓存投毒攻击（本章参考文献［3］对这一主题进行了详细评论），攻击者猜测 QueryID 和端口号（稍后将对此进行解释），然后将“响应”返回给热门查询（这也需要猜测）。因此，解析器的缓

㊀ 实际上，Punicode 的设计初衷是不仅可用于域标签，而且还可用于各种应用所使用的其他字符串，以便为国际化 URI 提供支持，将在本书后面对这一问题进行讨论。

㊁ 顺便说一下，大多数记录仍然未被认证。

存长时间（时长与 TTL 指定时间相同）被攻击者的 IP 地址“投毒”。现在，只要 QueryID 是渐变的，则对其进行猜测是相当容易的。（攻击者可以继续发出自己的查询来确定 QueryID。）只要端口号保持不变，则端口号也可以很容易被猜到。因此，虽然实现端口号随机化并非易事（将在本章稍后讨论 NAT 盒时进行说明），但是采用的对策仍是对 QueryID（使用伪随机数生成器）和端口号进行随机化。尽管如此，没有任何东西可以阻止攻击者随机使用 QueryID 发送诸多“响应”，据说这种攻击相当有效。

此外，直到 2008 年（IETF 发布 DNS 安全扩展标准很久以后），人们发现了一个新漏洞，其中整个区域可能被欺骗。具体来说，攻击者可以对域名服务器进行配置。该域名服务器声称在给定区域内它是经过授权的。（从本质上来讲，拥有这么做的能力不存在任何错误或危险，因为层次结构中级别更高的域名服务器不会指向它。）然后，攻击者发送“响应”与权限记录，将其委托给被欺骗区域内的授权服务器。实际上，授权服务器的名称是正确的，但胶水记录将提供攻击者的 IP 地址。

当然，存在着针对此类特定攻击的对策（因此，为响应任意给定攻击，人们需要不断更新域名服务器软件的补丁），同时，针对广义 DNS 安全问题，人们提出了多种相当简单的解决方案。RFC 3833 提供了一个完整的攻击目录。

也许最简单的解决方案是在客户端和域名服务器之间使用通过认证的信道，以便明确每个记录的来源。为此，可以由 IPSec（IP Security，IP 安全）信道来完成此工作。

另一种更为通用的解决方案是对每条 DNS 记录进行认证，以便客户端可以对其来源进行检查。1997 年，IETF 开发出一种解决方案，并发布详细说明域名系统安全扩展（DNSSEC）的首个标准（RFC 2065，现已作废）。围绕该标准的工作继续，当前的规范集（DNSSEC - bis）包含在 RFC 4033 ~4035 中。

该方案是使用公钥密码体制，并从根授权域名服务器开始，自上而下对信任链进行验证。实际上，DNSSEC 不仅提供了对记录来源的认证，而且还提供了其完整性保证（对客户端工作过程所做的任何修改）。要支持此功能，DNSSEC 将添加新的资源记录类型：

1）资源记录签名（Resource Record Signature，RRSIG）；

2）域名系统公钥（Domain Name System Public Key，DNSKEY）；

3）授权签名者（Delegation Signer，DS）；

4）下一代安全（Next Secure，NSEC）。

DNSSEC 还通过添加新标志来修改消息头。总之，这些修改以生成更大 DNS 响应消息作为结束。遗憾的是，该结果在本章稍后讨论的拒绝服务（Denial of Service，DoS）攻击中被利用。

新资源记录类型的用途如下所述。

RRSIG 记录存储了各个 DNS RRset 的数字签名。通过支持多个私钥对区域的数据进行签名（可能需要不同的算法），可能会使情况变得更为复杂。安全感知解析器的工作是学习区域信息的公钥。为了使此流程工作，解析器当然必须至少配置一个信任锚点。这可以通过配置信任锚点（充当用于构建针对 DNS 响应的认证链起点的公钥或其散列值）来实现。安全感知解析器可以从其配置或在 DNS 标准解析过程中学习锚点。为了达到在 DNS 标准解析过程中学习锚点的目标，我们引入了 DNSKEY RR。接着，安全感知解析器通过形成从新的 DNSKEY 到已知 DNSKEY 的认证链来对区域信息进行认证。为了使此流程正常发挥作用，解析器当然必须至少配置一个信任锚点。

DS RR 用于对跨区域边界的授权进行签名，它包含了授权子区域的公钥。并不是说复杂性总是对安全性有益，但 RFC 4033 指出，“DNSSEC 支持更加复杂的认证链，如在区域内 DNSKEY RR 的附加层对其他 DNS RR 进行签名。”

在默认操作模式下，认证链是从根区域到叶区域构建的，但 DNSSEC 支持的本地策略优先于默认模式。

迄今为止，已经描述了 DNSSEC 如何处理肯定响应。相比之下，NSEC RR 用于对否定响应进行签名。RFC 4033 解释道：“NSEC 记录支持安全感知解析器使用认证其他 DNS 回复的相同机制，来对名称或类型不存在的否定响应进行认证……NSEC 记录链对区域中域名之间的差距或‘真空区’进行显式描述，并列出存在于现有名称中的 RRset 类型”。

每条信任链都是如此，信任链与其最弱的环节一样健壮。如果这一信任链接恰恰位于根目录，则每条信任链都会很弱。因此，顶级域名服务器的安全性是需关注的主要问题。ICANN 通过编写用于签署根目录的脚本来解决这个问题，其中设计了包含响亮有趣名称的多个角色（如域名管理员、加密员、

安全控制员、内部见证人等)，以执行所需的步骤。这些步骤包括确保密钥存储的安全性最初为空㊀，将密钥生成设备带入房间，生成密钥并对密钥进行签名（生成证书），备份智能卡上的密钥，以及存储恢复密钥材料（在防篡改硬件安全模块中），然后将其放置在各种保险箱中。所有这一切都是在审计员面前完成的，且每一步骤都仔细记录在某个日志中。当然，所有参与者都可通过各自政府颁发的身份证件进行身份验证。不同房间拥有不同进入许可证。例如，不是每个人都可以进入安全房间。相反，在脚本编写结束或预定的休息时间之前，不允许任何人离开脚本编写室。当然，还存在着参与回收材料年度库存的流程。

遗憾的是，DNS 本身可以（在不破坏自身安全性的情况下）用于拒绝服务攻击，我们将其作为攻击家族的一部分将在下一节进行讨论。

DNS 对互联网的重要性不可低估。“互联网治理”的概念几乎意味着对 DNS 根服务器顶级域名空间的管理。20 世纪 90 年代，发生过一种情况，即某些 DNS 开创者对互联网治理有着截然不同的看法。在参考文献［4］这部经过深入研究写成的专著中，作者对已经发生的戏剧性事件进行了详细描述，其作者——哈佛大学和哥伦比亚大学法律学教授分别发表了他们对互联网治理问题的意见。

围绕互联网名称与号码分配机构（ICANN），存在诸多争议，但我们认为重要的是需指出 ICANN 已经实现了多少。以下是从本章参考文献［4］所引内容：“…［ICANN］分散了域名的销售和分销，导致注册价格大幅下降。建立了解决商标争议的有效机制，缓解了‘域名抢注’问题……命名和编号系统保持了足够的稳定性，人们很少担心互联网会崩溃。”谁还有其他问题吗?

5.2 防 火 墙

NIST（National Institute of Standards and Technology，国家标准与技术研究院）出版物[5]将防火墙定义为一种用于控制“... 处在不同安全态势的网络或主机之间的网络流量”的编程设备。

这一定义包含 3 个方面的内容。首先，防火墙不一定是以“盒子”的形态呈现出来——将在讨论网络功能虚拟化时进一步阐述该概念。其次，如图 5.8 所示，防火墙既可能位于两个网络之间，又可能

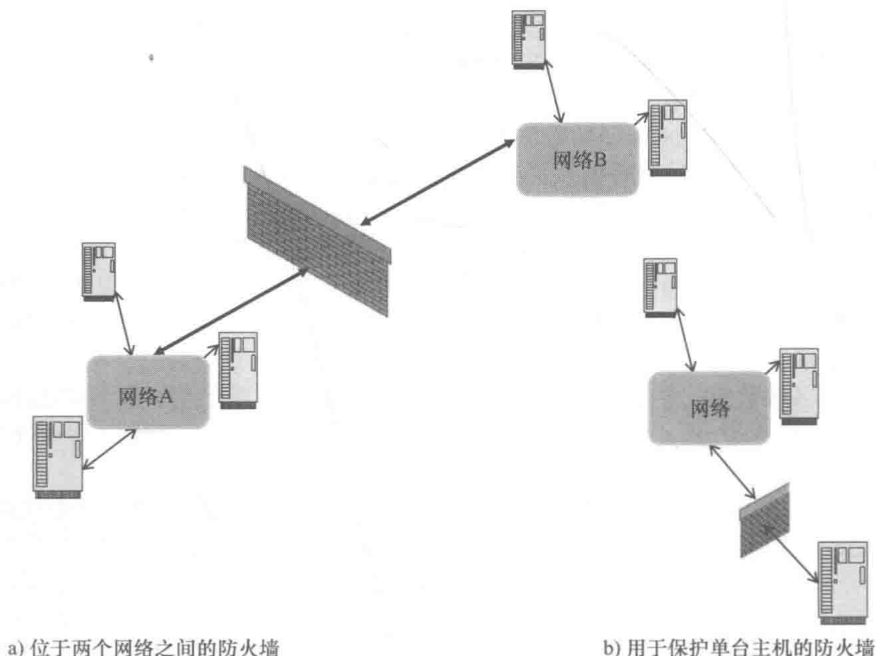


图 5.8　防火墙

㊀ 为了完成这一任务，密钥是必备的，这是设备的一部分。



位于单台主机和网络之间。(为此，防火墙可能由操作系统提供，或者正如稍后将要讨论的，以一种纯“软”的形式——由虚拟机管理程序来提供。）再次，防火墙可能不一定位于两个不同网络之间，分别采用不同安全策略的同一网络的不同部分（或区域）必须单独进行保护。

如图5.9所示，最后一点是至关重要的。通常情况下，“防火墙是第一道防线”，但是重要的问题是：谁是攻击者？例如，攻击[⊖]可能来自于机构内部，且第一道防线是根据各自策略来保护机构所在网络的每个部分。

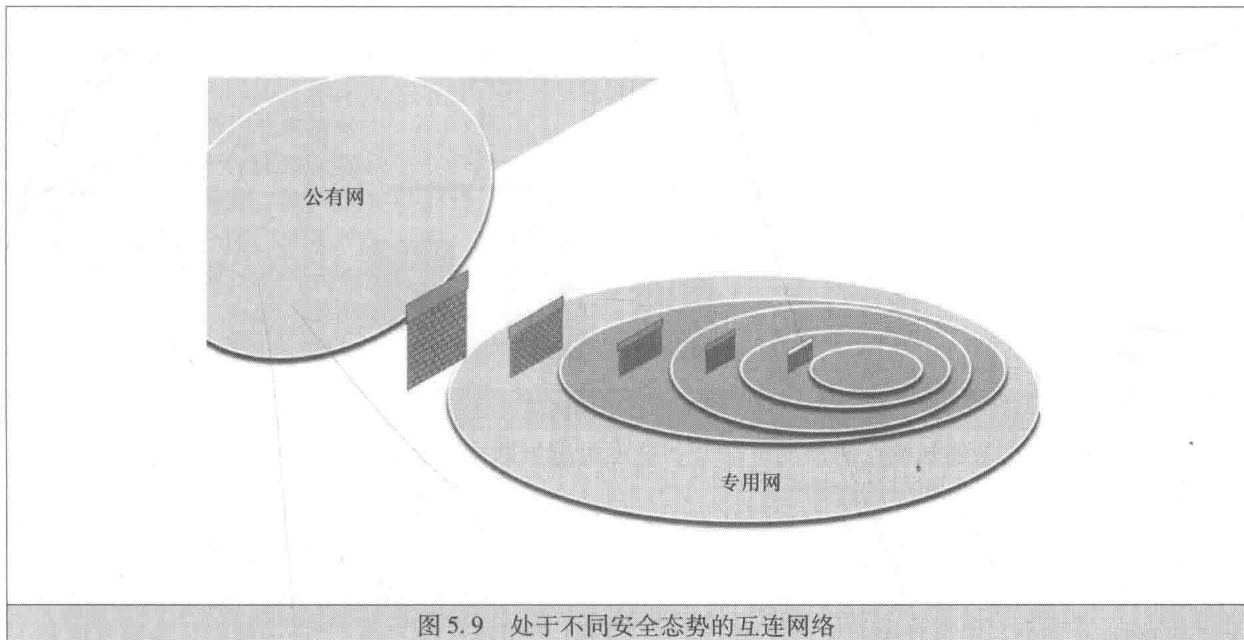


图5.9 处于不同安全态势的互连网络

防火墙不仅是为了阻止“恶意流量”进入。正如将要看到的，防火墙的另一个用途是防止非正当流量离开网络。正当的概念由安全策略进行定义，且会随时间和法律要求而发生变化。有趣的是，限制离开网络的流量不一定属于保密问题，正如父母对子女的不良行为负责一样，机构也需要对恶意流量源负责。

本章参考文献［6］对防火墙的早期历史进行了描述。根据本书前面章节所描述的内容，防火墙技术比数据通信技术出现得要晚得多——就像大多数安全技术发展的情况一样，防火墙技术的改进也是响应式的。第一种防火墙只是分离式局域网（LAN），且有意将其内置到路由器而不是第2层交换机中，以便完全终止流量广播。正如本章参考文献［6］所描述的那样，这些早期防火墙并没有将安全问题考虑在内。根据防火墙的原意（防止火焰从一个房间或建筑物蔓延到另一个房间或建筑物的手段)，这一思路只是为了防止局域网络问题的扩散，其中大部分问题当时是由配置错误引起的。

20世纪90年代初，作为网络安全引擎的防火墙开始变得活跃。最初，一些增加了执行用于限制某些目的地址和源地址的过滤规则的能力的路由器开始出现。同时，数字设备公司（Digital Equipment Corporation，DEC）和AT&T（American Telephone & Telegraph，美国电话电报公司）贝尔实验室的研究将包过滤与应用网关结合起来（见图5.10)。据报道，第一款商业防火墙已于1991年面市。

应用网关的功能远超过其名称所隐含的功能（采取一切措施确定的委婉说法）——在确保应用网关实际上是网络层设备一部分并继续检查网络和传输层数据报的同时，它还负责检查应用层的网络流量。当然，这一想法是不仅负责检查流量，而且还要通过限制流量（即丢弃可疑的数据包）来跟踪检查。当时，由防火墙开始执行的另一项基本任务就是创建日志。与潜在攻击有关的信息以前是且现在仍是非常宝贵的。

就互联网架构纯粹主义者而言，防火墙是令人深恶痛绝的。首先，应用网关开发公然违反分层原则；

⊖ 这里的“攻击”并不一定涉及旨在摧毁基础设施的暴力行为，它可能只是试图获取必须保密的信息。稍后将会看到一些常见威胁和已经实现的攻击。

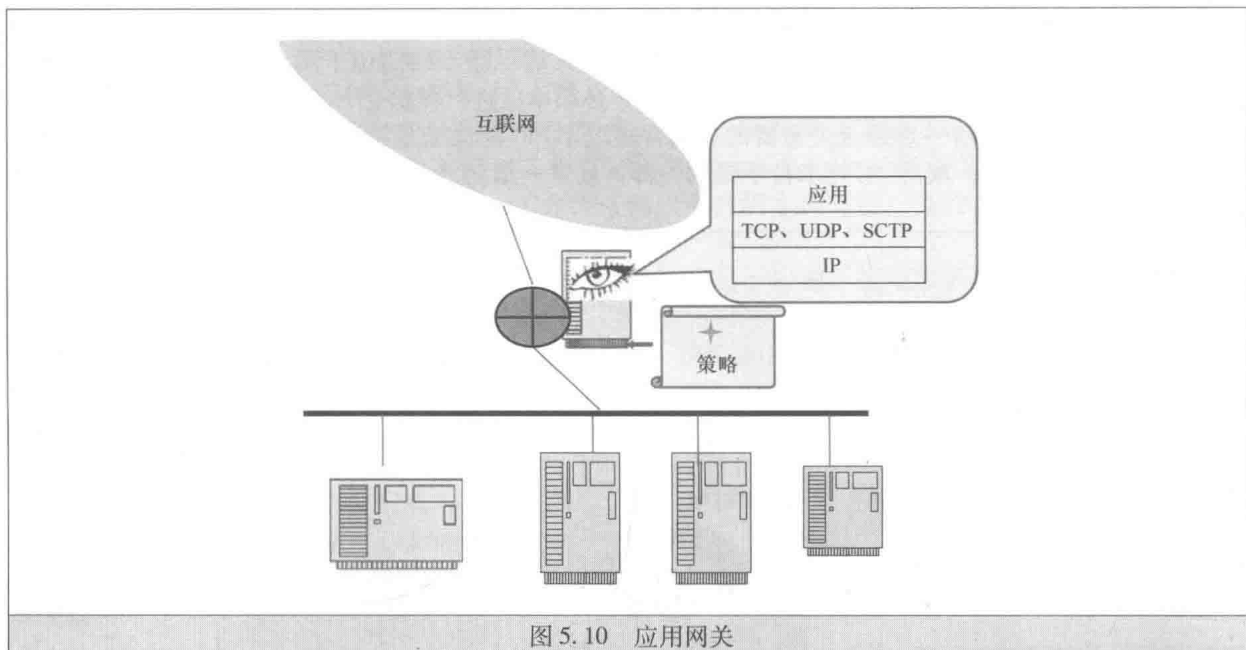


图 5.10　应用网关

其次，它导致通信在未通知端点的情况下断开，这不可能推断出为什么端点之间的流量丢失，因而可能会接受网络拥塞并采取相应的行动[㊀]。

然而，应用层防火墙（或应用网关）的引入不仅被网络管理员广泛接受，而且还被互联网研究人员和架构师广泛认可。本章参考文献［7］的作者之一[㊁]是一位狂热的互联网爱好者［后来，Steve Bellovin 博士入选互联网架构委员会（Internet Architecture Board，IAB），并领导了 IETF 研究领域］在《防火墙与 Internet 安全：击退狡猾的黑客（第 1 版）》（1994 年出版）中写道："……我们认为防火墙是一种重要工具，它可以最大限度地降低风险，同时提供网络连接的大部分优点（但不一定全部是优点）。"

需要注意的是，上述陈述出现在万维网、电子商务、互联网泡沫以及电信提供商、银行、报纸、广告商和犯罪分子针对 IP 网络的大规模运动之前。为了理解所涉及的威胁，需要研究一下 20 世纪 80 年代末前后的互联网应用。这些互联网应用包括：

1）文件传输；

2）电子邮件（顺便说一下，电子邮件几乎只能传输纯文本的附件，仍然采用 ASCII 编码，后来开始流行起来）；

3）远程电传（纯 ASCII 文本）终端服务（通过 Telnet 协议）；

4）远程登录包（包括远程 shell 执行）；

5）域名服务（采用 Finger 协议）——将其作为呈现事件包进行传递；

6）新闻组——一种公告栏讨论服务，它是互联网论坛的前身。

在今天看来，虽然这些服务过于简单和老套，但是仍然存在诸多安全问题。虽然还有些问题难于预防和取证，但是有它们是易于想到的（通过远程登录在未经授权的情况下访问远程计算机）。黑客滥用或多或少限于窃取软件，偶尔也会截获电子邮件，虽然后者导致的后果不像现在那么严重，因为当时公司间通信仍然以纸质备忘录、面对面的会议和电话（当时使用老式电话进行通信是非常安全的）的形式实现。

这些看似微不足道（但实际上尚未被利用的）的滥用手段对 1988 年互联网蠕虫攻击产生了巨大影响。在此次攻击中，康奈尔大学的研究生声称他只是试图确定互联网的规模，巧妙利用了泛在 Berkeley UNIX 版本中的 Sendmail 和 Finger 协议各自实现方案中的漏洞，以及因主机管理不善而导致的薄弱环节，用不断复制的程序来感染大量主机。

如果因计算错误而导致程序复制多次，启动新进程，并最终导致大量主机停摆不是事实，则运行

㊀ 通过采用更加糟糕的被称为反向代理的"异物"，可以缓解这一特殊效果，稍后将进行讨论。

㊁ Steve Bellovin 博士，当时是贝尔实验室的同事，现在是哥伦比亚大学的计算机科学专业教授和美国联邦贸易委员会（Federal Trade Commission，FTC）的首席技术官（Chief Technology Officer，CTO）。



该程序的过程被设计为（本来就应当如此）不被察觉的。本章参考文献［8］对该攻击进行了详细描述。

这一事件引起足够的轰动，以至于美国国会要求政府问责局（Government Accountability Office，GAO）调查此事。由此产生的报告使得阅读非常有趣。它也揭示了网络上挥之不去的损失估计的起源[㊀]。虽然报告中提出的具体建议标为“未实施”，但是对该事件的反应影响深远。

该事件导致的一个主要结果是，美国国防高级研究计划局（Defense Advanced Research Projects Agency，DARPA）将软件工程研究所（Software Engineering Institute，SEI）收入麾下，这是一家由联邦政府资助的卡内基－梅隆大学下属的研发中心，创立了 CERT®计划“以快速有效地协调安全应急领域专家之间的通信，以防止未来发生事件，并提高互联网社区的安全意识。”该项目在检测安全问题和分析产品漏洞方面仍然非常有效。

互联网蠕虫事件还通过提示安全产品将会有市场，来推动供应商公司中防火墙倡导者的事业发展。这一市场肯定会随着万维网的发展而增长，因为互联网上存在攻击。因此，人们已经开发出不同类型的防火墙及其混合体，来保护新应用，应对新威胁。

接下来，回顾一下引入特定防火墙技术的动机（在许多情况下，动机是应对攻击），并对防火墙技术进行简要描述。为此，讨论如下内容：

1）网络周边控制（包括 VPN）的基础知识；
2）无状态防火墙；
3）状态防火墙；
4）应用层防火墙。

5.2.1 网络边界控制 ★★★

拥有网络的每个机构都必须拥有由机构业务需求决定的安全策略，这是一种众所周知的做法。得到这种策略的形式化表达，并将其转换为计算机（通常情况）和防火墙（特殊情况）可以进行处理的东西，是正在进行的研发主题。为了撰写本章内容，仅列出可以转换为防火墙可用规则的相关问题。

第 1 个问题是可以访问网络资源（如 Web 服务器或电话网关服务器）的网络外部实体规范。对于每种资源来说，规范应当涉及该资源的各种网络实体、用户或用户组的集合。

第 2 个问题是访问任何给定资源的时限规范。一个简单的实例是连续时间段或一组连续时间段的规范。当资源访问需要收费时，问题将会变得更加复杂。

第 3 个问题是允许在资源上执行的操作规范[㊁]（如某个参数是只读的，或其值也可以进行更改的情形）。由于这些规范可能取决于特定用户（如允许受雇于网络所有者的网络管理员拥有远远高于客户的权限），因而我们将这些规范也指定为访问规则的一部分。

第 4 个问题是特定用户所需的认证类型规范。我们认为，这是迄今为止最复杂的课题，围绕这一课题，已经构建了身份管理学科和产业（将在本书后面进一步讨论）。

第 5 个问题是用于指定何种类型的异常（甚至例程）数据需要记入日志记录，何种行为将会导致即时告警的规范。

将指定访问网络内部资源的规范复制到访问网络外部资源中，上述 5 个问题变为 10 个问题。在本节前面提到，重要的是要记住，当将“保护”一词应用于防火墙时，它具有双重含义。阻止错误数据包进入网络只是问题的一部分，同样重要的问题是限制数据包输出：阻止某些数据包离开网络。（后一

㊀ 该报告宣称：“官方没有估计病毒到底感染了多少台计算机，部分原因是不存在负责获取这些信息的机构。据媒体报道，约有 6000 台计算机被感染。据报道，这一数字是根据麻省理工学院（Massachusetts Institute of Technology，MIT）的估计做出的。MIT 指出，10% 的计算机已被感染，将这一比例进行外推，即可估计得出受感染的计算机总数。但是，并非所有网站都拥有与麻省理工学院相同比例的存在漏洞的计算机。哈佛大学研究人员通过互联网询问用户，认为更准确的估计结果应当是 1000～3000 台计算机被感染。尝试估计与病毒有关的经济损失存在类似问题。受感染机器总数是未知的，且在每个站点病毒相关问题上花费的工作时间可能会有所不同。哈佛大学研究人员早些时候曾经提到，经济损失估计在 10 万～1000 万美元之间。”

㊁ 它听起来有些抽象，但是在 RESTful API 的情境中，将及时了解与这些操作有关的非常具体的实例。

种理念乍看起来显得特别奇怪，但考虑到机密数据从公司泄漏出来或其员工访问危险网站的情形，也就不足为奇了。另一个重要实例是：网络提供商有望通过使用未在各自网络内使用的源地址，来阻止IP流量以预防IP地址欺骗。）

然后，可以将策略规范转换为规则。防火墙必须对规则进行解释，这是逻辑编程学科所研究的一个值得正视的问题。这里，一个众所周知的问题是特征交互（涉及某一特征的规则可能与支持另一特征的规则相互矛盾，进而导致实时操作中的非预期后果）。预计策略描述语言将有助于明确指定这些规则，并确保它们能不按次序来应用这些规则。

这样，可以使用两套策略（从而对应两套规则）：用于控制入网流量准入的策略规定了入口过滤；用于指定哪些数据包不能离开由防火墙保护的网络的“出网”策略规定了出口过滤。这两套策略的区别如图5.11a所示。

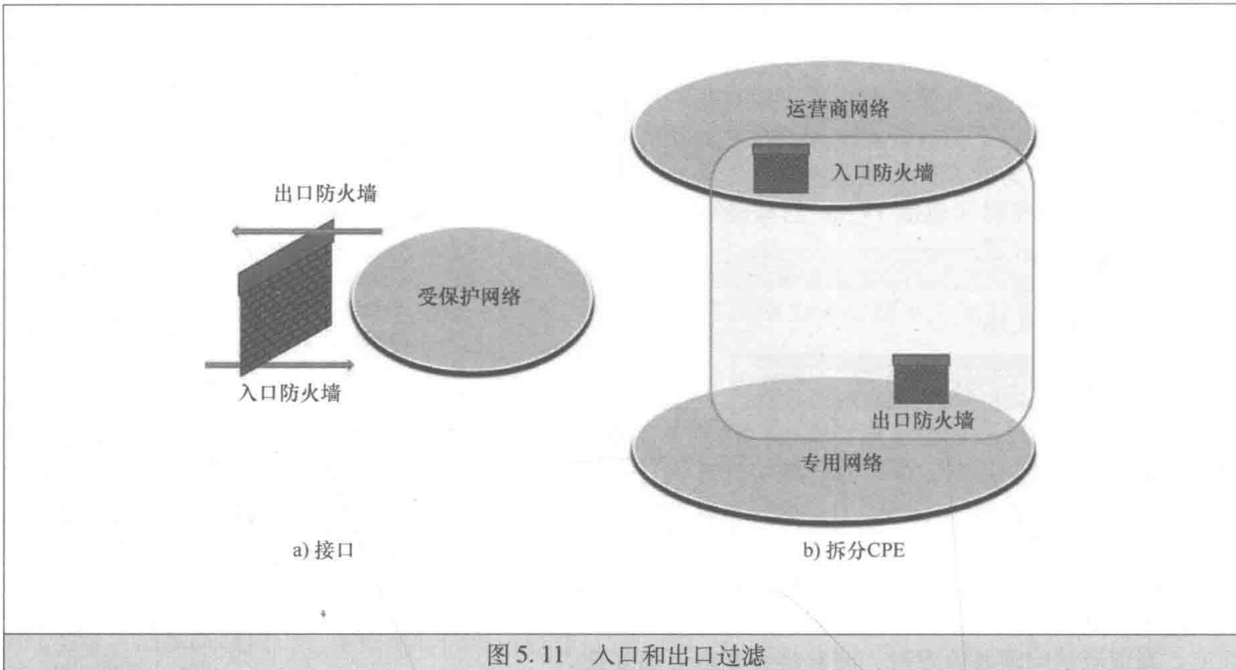


图5.11　入口和出口过滤

实际上，存在两种分别用于处理入网和出网流量的不同防火墙。当运营商在用户驻地设备（Customer Premises Equipment，CPE）中提供防火墙服务时，将其进行拆分是合理的，以便入口防火墙物理实体位于运营商网络内，如图5.11b所示。采用这种方式来访问网络不仅能够节省带宽，而且安全性也会得到增强，因为运营商更有能力缓解针对企业的某些拒绝服务攻击。这两点都适用于家庭网络，我们将于本书的后面在网络功能虚拟化的情境下对其进行讨论。

最后，当谈及“网络边界”时，应该回到第4章关于VPN（Virtual Private Network，虚拟专用网）的讨论，并回顾网络实际上可能是一个群岛的事实（见图4.2）。存在着两个问题：

1）就安全策略而言，群岛中的各岛必须彼此（以及与“大陆”）无法区分。

2）鉴于每个岛屿都被潜在的敌对水域包围，因而不存在一种类似防火墙的东西能将“我们与他们”区分开来。

第1个问题可通过每个岛屿确实受到一种或多种防火墙保护的事实来解决——每种防火墙针对一种外部连接。当然，必须在入口和出口访问方面实现相同的策略。第2个问题可以通过采用IPSec连接防火墙来解决，这样每条隧道都是安全的（独立于底层网络实现）。使用防火墙的3层VPN如图5.12所示。

现在，准备回顾提供服务的防火墙。在所有这些服务中，最简单、最古老的服务（无状态防火墙服务）只能根据源IP地址和目标IP地址来过滤IP数据包，有时也会根据协议类型来过滤IP数据包。



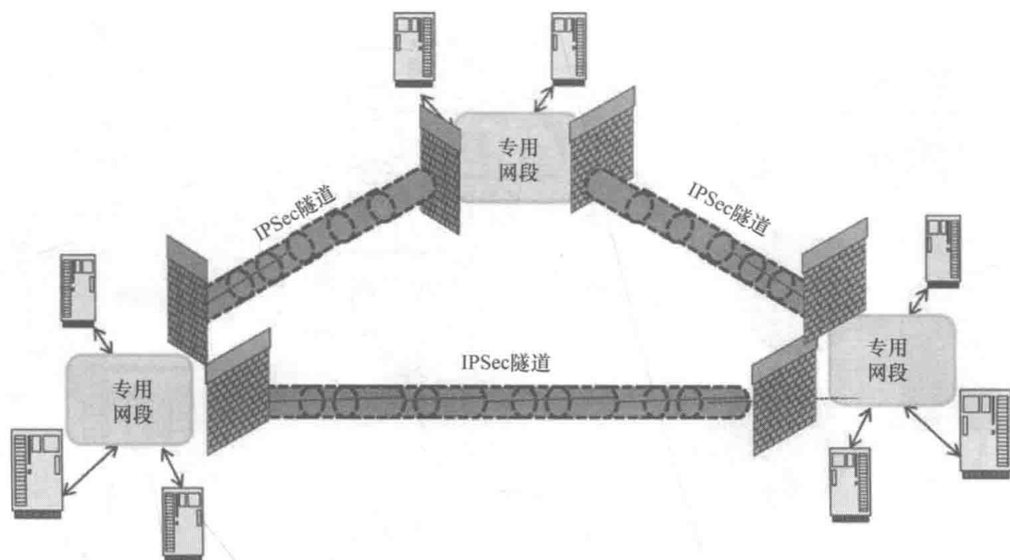


图 5.12 使用防火墙的3层VPN

5.2.2 无状态防火墙 ★★★

为了举出一个用于说明为什么要根据协议号（而不仅仅是IP地址）来限制流量的实例，需要简要地介绍另一个协议——互联网控制消息协议（Internet Control Message Protocol，ICMP），该协议在RFC 792中定义，并在其他几份RFC中进行了更新，特别是在IPv6方面。ICMP拥有自己的协议号（它实际上是1），然而，由于ICMP不携带任何的端到端有效载荷，因而它不属于传输协议。虽然ICMP在诸如ping（该应用允许确定主机是否存在）和traceroute（该应用能够发现指向主机路径上的所有路由器）等应用中得到充分利用，但是其工作主要是为了在路由器之间传递可达性问题。遗憾的是，ICMP也可用于发起一系列DoS攻击，从1997年的smurf攻击开始（据称是由一名高中学生发起的）。

如图5.13所示，攻击针对IP地址已知的给定服务器。攻击者在发送到广播地址块的ICMP回应请求中对该地址进行欺骗。当请求终止于局域网时，路由器强制传输请求，有助于将其转换为链路层广播请求（需要主机响应的请求）。参与的网络充当的是放大器角色。当放大器中的所有这些主机都向服务器“响应”时，后者很快就被处理流量所淹没，导致对其他任何东西均是一无是处。大量的数据流也可能压垮服务器所依附的网络。为此，CERT（Computer Emergency Readiness Team，计算机应急响应小组）报告CA－1995.01指出：“这种攻击的中间人和受害者在其内部网络或它与互联网的连接中，都可能受到网络性能恶化的影响……对于为中间人或受害者提供服务的中小型ISP（Internet Service Provider，互联网服务提供商）来说，足够的流量可能会导致性能严重恶化。大型ISP可能会观察到主干网性能恶化和对等饱和。”

对smurf攻击的反应会导致路由器配置的变化，以及对Internet标准进行修改，不要求（默认设置）指向广播地址的数据包实际转发给广播地址。在不采取这些措施的情况下，由运营商负责的IP分组入口过滤能够防止运营商成为攻击的放大器，其中IP分组源地址与发布源地址的网络地址不匹配。这是由RFC 2827规定的，该文档拥有当前最佳惯例的美称。我们还注意到，通过发送端运营商的出口防火墙来防止欺骗是所有解决方案中最简单的方案（虽然它无法阻止对该运营商网络内主机的攻击，这反过来要求所有出口防火墙实现这一功能）。采取的另一种措施是由网络管理员对ICMP流量进行限制。这应当能够解释为什么防火墙查看字段是可取的。

除了ICMP的使用是可有可无的以外，smurfing不是万能的，它只是一种承载攻击的手段。恶意“请求”的放大（通过广播）和多台“响应”请求的服务器的反应的思路是足够抽象、足够强大的，可以成为创建大量拒绝服务攻击的一种方法。

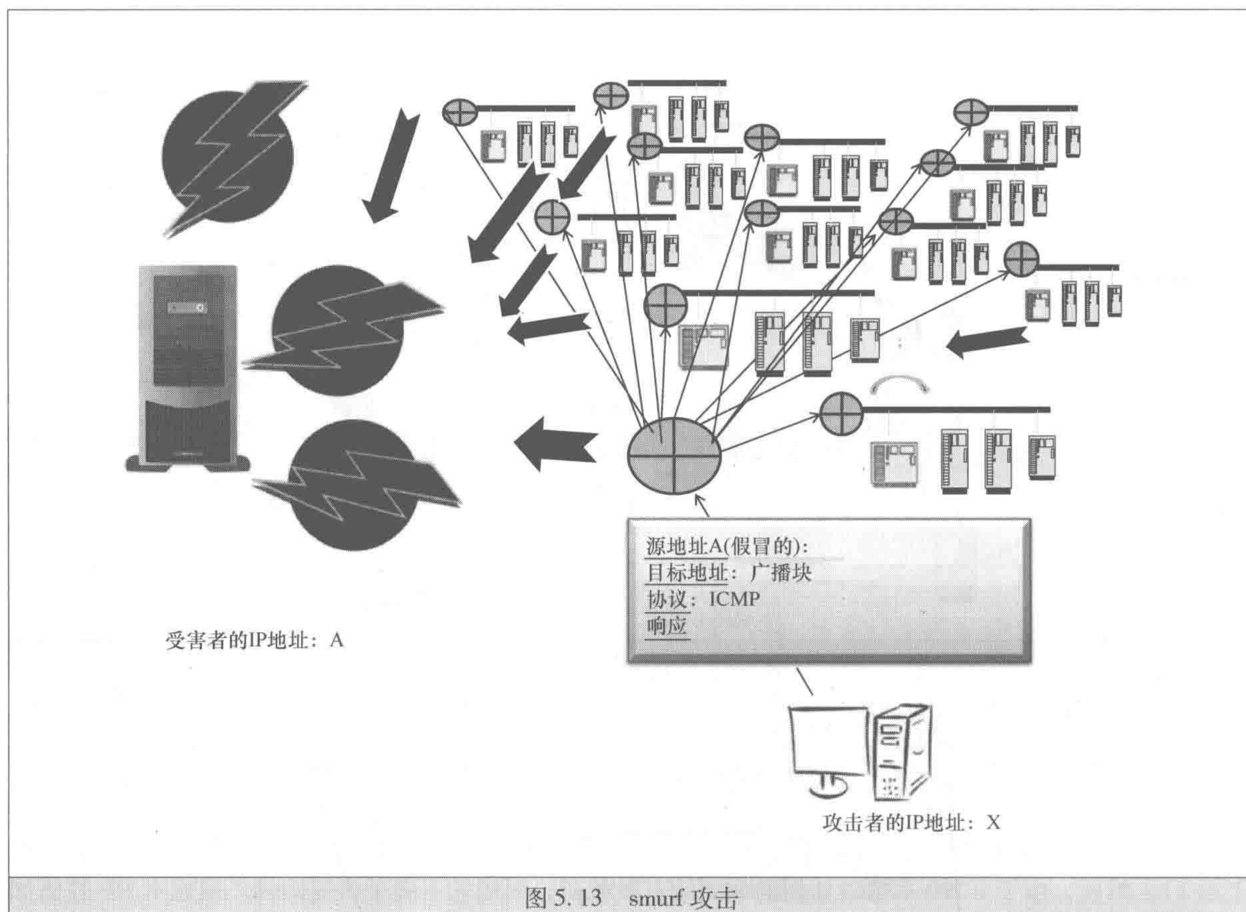


图 5.13 smurf 攻击

乔治·波利亚（George Pólya）在参考文献［10］中有句名言："方法和设备的区别何在？方法是一种你可以使用两次的设备。"当然，使用同一方法编写 smurf. c 程序的人，后来又编写了 fraggle. c 程序（一种比前者更大的程序，但仍然只有 136 行长）。那种攻击完全没有使用 ICMP。相反，它将 UDP 流量用于与测试相关的端口 7 和 19。端口 7 连接到回应服务；端口 19 连接到 chargen（字符发生器协议）服务，以随机字符串的形式进行响应。

fraggle 攻击的效果与 smurf 攻击类似。显然，将在所有主机上禁用这些服务作为默认设置，可能会阻止这种特定攻击，但这样的结果过多，会超出系统管理员预期。除非攻击在到达防火墙之前已经阻塞网络（或只是阻塞链路），否则无论做什么都是徒劳。另外，防火墙可以通过消除针对这些端口的流量（越接近网络边界越好）和记录相应活动来发挥作用（日志分析器可以向合适的响应团队发出告警）。更为重要的是，防火墙在问题的根源上始终有效。在造成任何伤害之前，可以消除使用欺骗地址的数据包。遗憾的是，放大攻击并没有如前所述的那样结束，他们似乎无止境无期限。

如前所述，DNS 成为此类攻击的目标。在 DNS 放大攻击（见图 5.14）中，攻击者会欺骗用于打开递归解析器的请求[⊖]，这反过来又使用 DNS 响应流量对目标主机（其 IP 地址被欺骗）进行泛洪。CERT 通报 TA13-088A 对 DNS 放大攻击进行了详细描述。需要注意的是，构建请求以提供所有可能的区域信息（通过使用 ANY 类型）。可悲的是，DNSSEC 使事情变得更加糟糕，因为由此形成的响应远远大于因签名而不使用 DNSSEC 的响应。最后，可使用僵尸网络（可能位于不同的网络中）进一步放大效果，从而导致大规模分布式拒绝服务（Distributed Denial of Service，DDoS）攻击。

此类攻击的变体也可能涉及无法提供递归解析功能的授权域名服务器，但在这种情况下，可以通过限制响应速率来缓解攻击。

⊖ 开放意味着向公众开放。通常，人们使用服务提供商网络内的递归解析器，且服务提供商网络只允许由该网络内的客户进行访问，因而可以阻止任何恶意活动的尝试。开放解析器项目负责跟踪开放递归解析器。

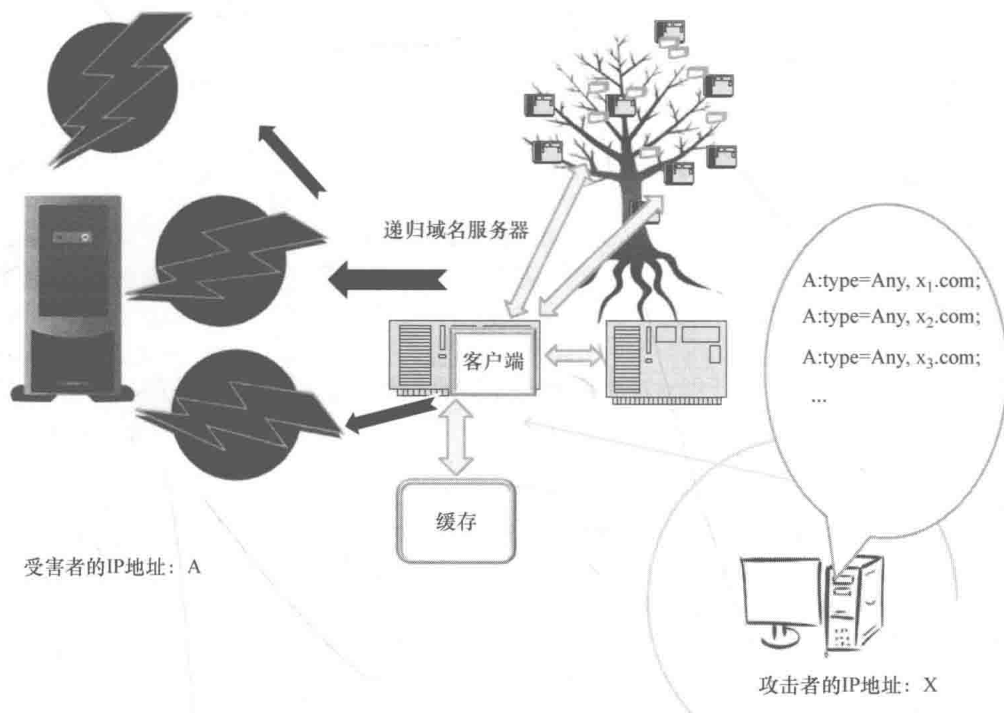


图 5.14 DNS 反射放大攻击

同时，根据 IETF 标准，源 IP 地址验证是防止此类攻击的最佳保护措施，但这需要所有提供商的合作——绝不允许使用欺骗地址的 IP 数据包离开网络。这里，出口防火墙是不可或缺的。为了对抗 DNS 反射放大攻击，NIST 建议防火墙在出接口上跟踪 DNS 请求，从不允许对进入网络的不存在查询进行"响应"。

我们将仅提供包过滤的防火墙称为无状态防火墙，因为与实际上遵从传输层连接状态的状态防火墙相比，无状态防火墙可以在不考虑总体流量的情境下，对每个数据包进行检查。现在已经有机会使用状态防火墙：人们可能会认为，前一段 NIST 建议中的实现方案实际上需要一种应用层状态防火墙，因为它涉及对所有未完成 DNS 查询的跟踪。

5.2.3 状态防火墙 ★★★

开发此类防火墙的最初动机是缓解 SYN 攻击，本章参考文献［11］对此类攻击进行了详细描述，其作者还编写了 IETF 于 2007 年发布的（信息类）RFC 4987。该攻击利用了 TCP（Transfer Control Protocol，传输控制协议）的典型实现方案（具体来说，TCP 连接建立阶段）。

这里，理解发生了什么与理解为什么发生同样重要㊀。连接建立机制的相对复杂性是由同步发送方和接收方的初始序列号值的必要性引起的。原始的 TCP 规范解释道："对于即将建立或初始化的关联，两种 TCP 的序列号必须在彼此的初始序列号上实现同步。因此，该解决方案需要一种用于选择初始序列号（Initial Sequence Number，ISN）的适当机制，以及用于交换 ISN 的简单握手协议。'三次握手'是非常必要的，因为序列号与网络中的全局时钟并不关联，且 TCP 的序列号可能具有不同的 ISN 选择机制。SYN 的第一个接收方无法获知数据包是否为延迟的旧数据包，除非它记住并非总是可能的连接上使用的最后一个序列号，因而它必须要求发送方验证这个 SYN"。

图 5.15 描述了成功建立连接的交换机。接收器和响应器的状态分别保持在三位（S_1，S_2，R）中，表示是否满足如下条件：

㊀ 这样做是惟恐读者认为可以通过更改 TCP 来解决问题。



图 5.15　TCP 连接建立（引自 RFC 675）

1）发送具有序列号（$S_1=1$）的初始 SYN；

2）通过发回 ACK 参数序列号来由另一方确认，原始值递增 1（$S_2=1$）；

3）接收到 SYN（$R=1$）。

当双方达到状态（1，1，1）时，连接建立完毕。

在不成功的信息交换中，确认消息永远无法到达。在这种情况下，参与各方希望在相应定时器到期前返回到状态（0，0，0）。

接下来，将重点关注响应主机（为了达到我们的目的，将其看作一台承载攻击的服务器）。接收到第一条 SYN 消息［即向状态（0，0，1）的过渡］的主机分配对应的存储器（通常为阵列单元）。当然，存储块需要保存，直到连接返回到状态（0，0，0）。因此，存储块需要保存到超时。

如果太多 SYN 数据包在足够短的时间内到达，则响应者将耗尽内存，这反过来将导致系统崩溃或至少无法打开任何新连接。为了绑定服务器，攻击者可以做到这一点（而不需要对最后的 ACK 消息采取进一步行动）。RFC 4987 引用了一条有趣的记录，即本章参考文献［7］的作者设想了此类攻击，并在 1994 年版中写了一段文字，但后来决定将其删除㊀。

1996 年，被 CERT 称为“地下杂志”的期刊对 SYN 泛洪攻击首次进行了描述。随后，CERT 发布了 CA－1996－21 报告。由于攻击能采用欺骗 IP 地址的手段，以分布式方法有效实施，因而互联网社区的后续行动具有过滤的需求——重点强调了提供商入口防火墙的行动需求。遗憾的是，这一措施本身是不够的，因为一旦被劫持成为僵尸网络的一部分，则这些攻击可能会由看似合法的主机实施。

其他提出的措施只能通过改变操作系统内核（TCP 实现的地方）来实现。这一特征可提供配置网络连接数（半开连接数）的能力，使得一旦达到极限值，则关闭半开连接，释放其内存，并可能将这一动作与忽略 SYN 请求结合起来。大型服务器的网络连接数可能会增加，且定时器值也可以进行修改。

虽然，问题是由早期内存分配引起的，但是潜在解决方案必须做一些事情来推迟这一行动。一种高效的技术（在后来的防火墙实现中也暗示了这一点）是 SYN 缓存。在这种技术中，连接建立之前不对实际内存进行分配。当然，这里的问题在于它需要在主机的操作系统中进行（非同寻常）的更改，但是存在多种操作系统。

另一种技术被称为 SYN Cookies，它使用加密体制和预留给序列号的位，来对半开连接状态进行编码。当连接完成时，可以在新分配的存储器插槽中重建状态。这种技术的缺点是某些 TCP 解决方案在使用时变得不再可用（因为数据位被占用）。正如本章参考文献［11］所述，可以将这些技术组合到一

㊀ Bill Rosen 的原始叙述指出：“这种被称为 SYN 泛洪的技术可追溯到 1984 年（原文如此）。如果不是更早，则该技术最早就是出现在 Bill Cheswick 和 Steve Bellovin 出版的图书《防火墙和互联网安全：击退狡猾的黑客》中。‘我们在书中有一段话，“Cheswick 从贝尔实验室的办公室告诉我，我们删除的原因是由于我们知道无法解决这一问题。现在，我们感到非常抱歉。当时我们应当把它写进书中。’”

个强大的混合体中。

状态防火墙解决方案的实例如图 5.16 所示。

防火墙（大概本身由像 SYN 缓存或 SYN Cookies 这样的技术来保护）会响应 TCP 连接的发起方，尝试在响应方知道请求相关信息之前建立自己的连接。如果成功，则它建立与响应方的另一条连接，并从此中继发起方和响应方之间的数据，而发起方和响应方都不知道中间的代理（或中间盒）。

正如要看到的，可以将代理配置带到应用协议中，但是状态防火墙不一定是代理。在一个不太极端的例子中，防火墙可能只能跟踪所有连接（通过维护所有五元组 < 源 IP 地址，目标 IP 地址，协议，源端口，目标端口 > 的列表），而不是建立两个端点。

此类防火墙仍然比无状态防火墙的功能强大许多。例如，此类防火墙可以支持一种任何会话的持续时间不能超过指定的时间段［如果考虑根据连接持续时间进行收费（特别是在使用预付卡时），则这是一种重要策略］。

对开放式连接的跟踪与 NAT 功能不谋而合，这一点将在本章的下一节中进行探讨。

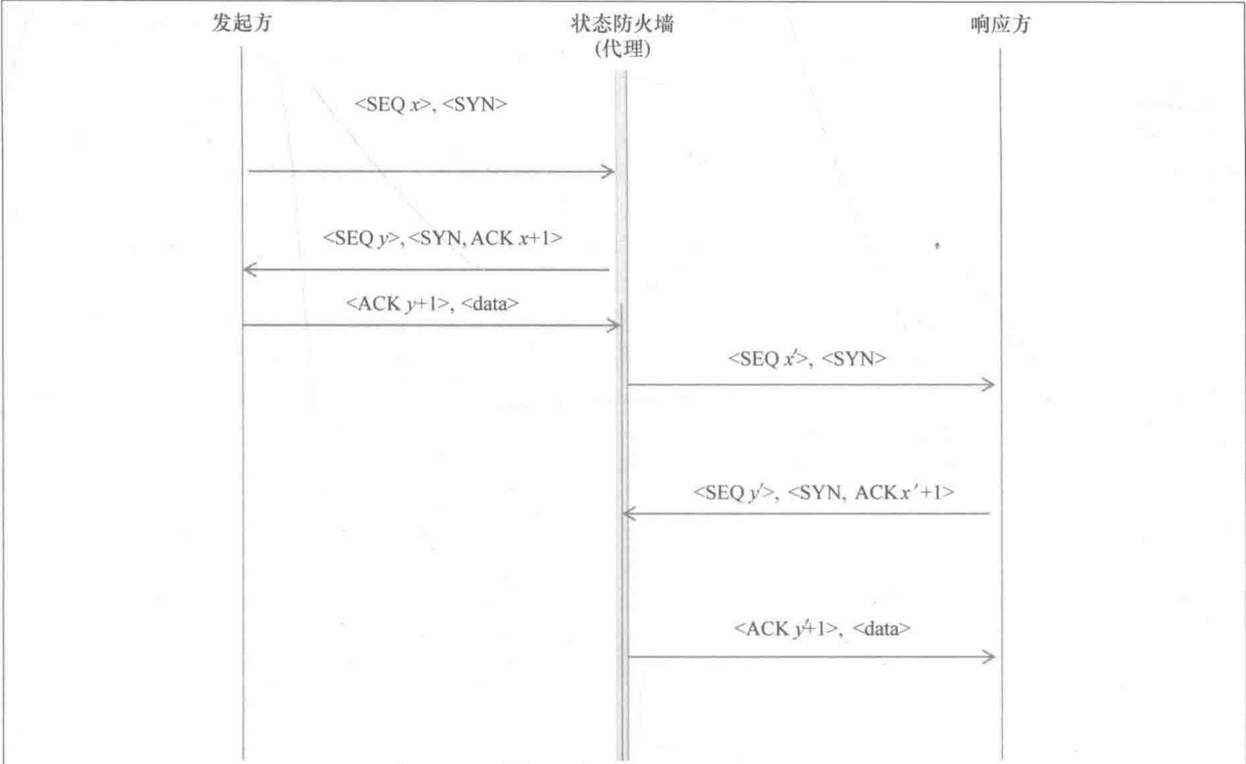


图 5.16 状态防火墙（TCP 连接建立的实例）

5.2.4 应用层防火墙 ★★★

在 4 月 1 日发布一份荒谬、幽默的 RFC 已经成为 IETF 的传统⊖。为此，由 Mark Gaynor Scott Bradner 于 2001 年撰写的 RFC 3093 在不违反防火墙安全模式的情况下，提出了“防火墙增强协议（Firewall Enhancement Protocol，FEP）”来支持创新。更具体地说，“建议”是“在超文本传输协议（Hyper Text Transfer Protocol，HTTP）上对任何应用层传输控制协议/用户数据报协议（TCP/UDP）数据包进行分层，因为 HTTP 数据包通常能够通过防火墙。”这确实是一个非常有趣的玩笑！当时，没有什么比在面向事务的应用协议上传送 IP 数据报看起来更加荒诞不经的了，该协议旨在检索万维网所依赖的小文件（称为网页）。

本书稍后将对 HTTP 进行讨论。这里观察到（不必大惊小怪），2001 年荒谬事三四年后竟然成为事

⊖ 也许其中最著名的就是 1990 年 4 月 1 日由 D. Waitzman 发布的 RFC 1149——“信鸽 IP 数据报传输标准”。改编过程可描述如下：“将 IP 数据报采用十六进制打印在纸卷轴上，每 8 位由白色和黑色分开。将纸卷轴缠绕在信鸽的腿上，并用胶带固定好数据报的边缘。带宽不能超过腿长。”根据经常引用的 IETF 轶事，当某个网络提供商在请求注解文档中包含符合 RFC 1140 的问题时，一些供应商在其响应中声称遵守此类规定。

实上的规范。事实上，HTTP 开始用作远程过程调用的传输协议，甚至用于视频流。原因在于 RFC 3093 中提到的——HTTP 可以不受阻碍地穿过防火墙。此外，某些对等应用（包括即时通信）使用端口 80——该端口通常预留给 HTTP 使用。当然，所有这一切意味着 HTTP 不能再像从前那样穿过防火墙：必须对“HTTP 消息”的内容进行分析。

活动内容（如 Java 程序）对恶意软件造成了重大威胁，因而需要对其进行检查。但即使将“正常”的 HTTP 方法用于不良目的（如引起缓冲区溢出攻击），且为了处理这一问题，至少验证输入参数长度是至关重要的。

因此，应用层防火墙已构建用于 HTTP、电子邮件、与 SQL（Structured Query Language，结构化查询语言）服务器和 IP 语音（Voice over IP，VoIP）应用的交互。即使是一种看似无害的数据结构规范语言——可扩展标记语言（Extensible Markup Language，XML）也提出了对 XML 防火墙的需求，主要是要了解写入其中的服务规范。例如，已经开发了基于 XML 的网络服务描述语言（Web Services Description Language，WSDL）来描述由站点提供的服务程序（将被远程执行）。

将应用层协议有效载荷的分析称为深度包检测。但是，应用层防火墙远不止检查有效载荷一项要求。实际上，在很多情况下，为了理解有效载荷，需要知道协议的特定状态。因此，应用层防火墙通常是有状态的。在极端版本中，应用层防火墙可充当代理的角色，因为它可以终止相关协议，并使用它所保护的防火墙后面的实体来重新启动其他实例。这种情形与图 5.16 中描述的情形类似。事情可能变得相当复杂，且这种复杂性可能会被（甚至已经被）攻击者利用。导致控制主机的攻击现在可能会发展为对防火墙的控制，这不足为奇，因为防火墙正在执行受保护主机应该执行的协议，且查看受保护主机可能查看的数据。正如尼采（Nietzsche）在本章参考文献［12］中所警告的：“与恶龙缠斗过久，自身亦成为恶龙；凝视深渊过久，深渊回以凝视。”

企业将防火墙部署到分离区域已经变得非常常见（有时甚至部署在单一机构中）。这一思路是在防火墙之后建立一种完全受保护的内部网络（通常称为可信区），同时还可以维护另一种称为隔离区（Demilitarized Zone，DMZ）的网络，该隔离区根据不同的策略运行。DMZ 的典型用例是托管企业的公共 Web 服务器[⊖]。DMZ 在不同策略下运行，因而它与外部世界和保护区分开。

可以使用至少拥有 3 个网络接口的单一防火墙来实现分区，如图 5.17a 所示。一个接口连接到外部

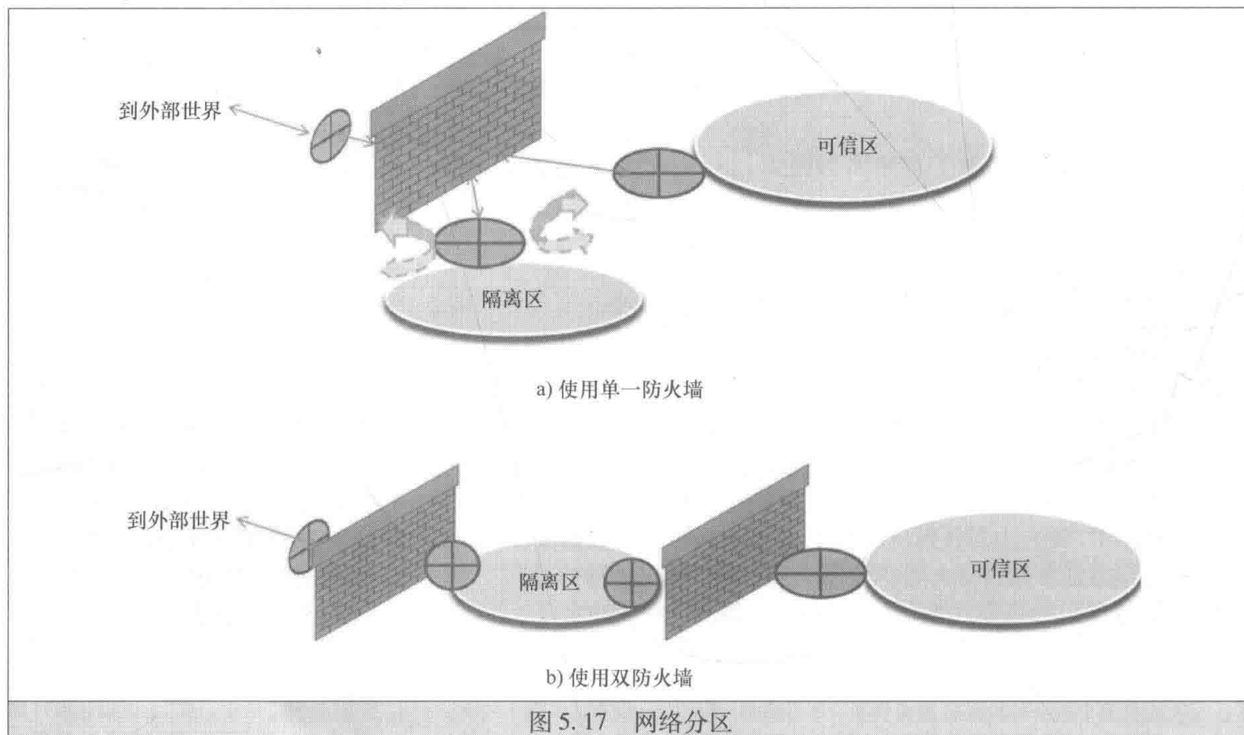


图 5.17　网络分区

⊖ 将在下一节进一步澄清为什么这是必要的。

世界，另一个接口连接到 DMZ（这样来自外部世界的数据流、输出到外部世界的数据流、来自保护区的数据流以及输出到保护区的数据流都通过该接口），第 3 个接口连接到可信区。这种配置是成本最低的，但它会带来危险（考虑到配置错误或单点故障的情形）。

图 5. 17b 中的双防火墙配置成本更高，但也更加安全。

正如将要描述网络地址转换（NAT）的功能一样，注意到，虽然它与防火墙功能不同，但它也通常作为防火墙的一部分来实现。

5. 3 NAT 盒

使用 32 位 IP 地址字段，全球大约有 43 亿个 IP 地址——显然这不足以为每个人的计算机、手机、洗衣机、牙签之类的东西分配地址。早在 20 世纪 90 年代，人们就已经明白了这一点。虽然为计算机之外的东西分配 IP 地址的想法出现得晚一些——为电话分配 IP 地址的想法出现在 1997 年左右；为洗衣机分配 IP 地址并将其作为智能电网计划一部分的想法出现在 2003 年左右；为牙签分配 IP 地址……好吧，我们推测，牙签是“物联网（Internet of Things，IoT）”中的“东西”。

IPv4 仍在广泛使用，但需要探讨解决 IPv4 地址空间不足的方案。这一想法是按照一种智能方案来重用网络中的现有 IP 地址池。

在这一方案中，首先将 IP 地址空间分为私有地址和公网地址两部分，其中私有地址仅在网络内部使用，即用于网络内通信。其次，对于每个输出的 IP 包，其源 IP 地址被更改为公网 IP 地址。这种地址替换正是 NAT 盒的功能。

接着，NAT 盒可以为通过它的所有数据包分配相同的公网地址。因此，如果网络中有 n 个实体，所有实体只共享一个 IP 地址，则该 IP 地址可用于外部通信。该方案不仅支持 IP 地址重用，而且还可以对 NAT 盒[㊀]后面的网络地址进行匿名处理，如图 5. 18 所示。

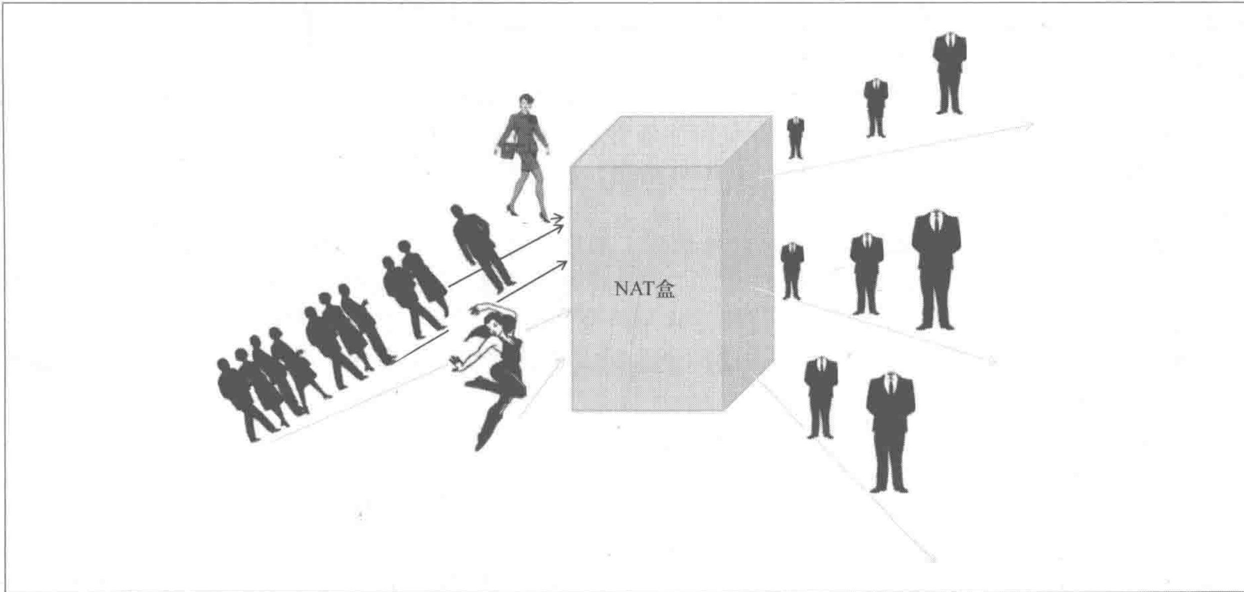


图 5. 18 NAT 简图

当然，如果没有引入新的（严重）问题，则人们不会放弃这一技术。网络隐私问题是因需要将输入分组的目的地 IP 地址转换成正确的私有 IP 地址而导致的。这一问题可以采用一种相当简单的方式来解决，但解决方案本身就是有问题的：正如将看到的，任何网络外部实体都不能发起与网络中任何实体的通信。这违反了互联网的主要原则。事实上，将会看到这一方案打破了一些互联网原则。但是，

㊀ 虽然此功能绝对不能代替安全措施，但它确实有助于提高安全性。因此，该功能通常被称为“网络隐私”。它支持网络运营商（无论是在企业界还是在服务提供商领域）使内部实体仅在需要时才能寻址。因此，一些实体对外部网络来说不可寻址。此外，基本的竞争信息（内部网络结构和设备部署号码）也会受到保护。

那些认为对NAT深恶痛绝的人也已经学会（并教导他人）如何与NAT友好相处。随着NAT部署进一步推进，以及互联网主机数量的进一步增长和IP地址空间的耗减，新问题及其解决方案数量不断增加。然而，NAT盒正在演进，同时仍然服务于上述两个主要用途：①稀缺IP地址的有效管理；②内部网络结构的模糊处理。

同时，后者的重要性不可低估。虽然在前面的脚注中提到了这一点，但是需要对这一问题进行扩展。首先，在企业网络（如银行网络或更为明显的实例——军事网络）中，由于众所周知的原因，存在着一些对外部世界不可见的主机。事实上，防火墙可以在这里发挥作用，但主要问题是这些主机无法从互联网地址空间进行寻址。其次，考虑到互联网的设计方式，主机和路由器都可以通过IP地址来访问。当涉及运营商网络时，拥有通过网络管理接口访问路由器的能力是一件非常危险的事情（这一问题不应该与直接连接对等路由器的问题混淆起来，后者是在链路层实现的）。在很大程度上，PSTN（Public Switched Telephone Network，公共交换电话网络）的安全性比较高，因为其交换机和其他网络元件是不可寻址的，因而无法从外面进行访问。互联网服务提供商意识到，他们最好遵循NAT支持的相同模型。

随着家庭网络（该网络包含一个部署在家庭网关中的NAT功能实体）的发展，结合企业网络的发展，运营商需要引入NAT上的NAT，通常称其为运营商级NAT或大规模NAT。

在本节的剩余篇幅，将讨论：

1）私有IP地址的分配；

2）NAT盒的架构与操作；

3）NAT存在的情况下支持（诸多现有互联网协议）运行的协议——交互式连接建立（Interactive Connectivity Establishment，ICE）协议、NAT会话穿透效用（Session Traversal Utilities for NAT，STUN）协议、使用中继NAT穿透（Traversal Using Relay NAT，TURN）协议；

4）大型NAT。

5.3.1 私有IP地址分配 ★★★

1994年，RFC 1597首次发布了关于私有IP地址分配的权威指南，它是由代表互联网行业3个重要组成部分——软件供应商、企业网络和互联网注册管理机构的工程师[⊖]合作完成的。解释引入私有地址空间的优点（IP地址保护和操作灵活性），RFC指出，私有IP地址的（不协调）使用已经发生，且警告了不良后果：

“由于各种原因，互联网已经遇到过这种情形，即没有连接到互联网的企业为其主机赋予IP地址空间，而该地址空间并非IANA分配给它的。在某些情况下，这一地址空间已经被分配给其他企业。当这种企业以后连接到互联网时，它可能会产生非常严重的问题，因为IP路由存在歧义寻址的情况，无法提供正确的操作。使用私有地址空间为这种企业提供了一种安全的选择，避免一旦需要外部连接可能发生冲突的情况。”

有趣的是，6年后，当技术人员负责为2001年3月召开的IETF会议（由朗讯科技发起的21世纪首届IETF会议）构建IETF网络时，亲眼见证了这种冲突。贝尔实验室和朗讯科技业务部门提供了大量专业知识，且该网络已经通过了贝尔实验室的所有现场测试。经过设计和构建，该网络可容纳3000多台主机，包括有线和无线连接。然而，在部署过程中，出现了一种奇怪的现象：即使网络在两大ISP上提供多宿主服务以确保可靠性，但许多参与者还是根本没有外部连接。

下面介绍当时空间发生了什么事。对于内部IETF网络来说，朗讯科技使用其私有空间的IP地址，但相关IP地址的特定块是1996年从AT&T开始实施后继承下来的。有些特定块明显存在问题，因为AT&T员工忘记将这些地址从其空间中删除。幸运的是，这一问题在几个小时内就被诊断出来，但我们会永远铭记它所引起的焦虑。

⊖ 其中包括BGP（Border Gateway Protocol，边界网关协议）的发明人——Yakov Rechter，他当时在IBM（International Business Machines Corporation，国际商业机器公司）工作；IETF安全专家Bob Moskowitz后来联合主持IPSec工作组，也参与了所有安全协议的开发，并负责克莱斯勒公司的安全；Daniel Karrenberg和Geert Jan de Groot在欧洲网络IP号码协调中心，RIPE是欧洲、中东和中亚的区域互联网注册机构；Eliot Lear在美国硅图公司（Silicon Graphics，SGI）工作。

1996年，（信息类）RFC 1597被拥有当时最佳惯例美誉的RFC 1918[⊖]所替代，该协议目前仍然有效。按照说明，IANA（互联网号码分配局）已经为私有网络预留了以下3类IP地址空间块：

1）10.0.0.0~10.255.255.255（10/8前缀）；

2）172.16.0.0~172.31.255.255（172.16/12前缀）；

3）192.168.0.0~192.168.255.255（192.168/16前缀）。

这相当于分别分配单个A类网络地址、16个B类连续网络地址和256个C类连续网络地址。

任何实体都可以在其域中使用这些网络地址，而不向IANA或任何其他人请求："决定使用本文档中定义地址空间中IP地址的企业可以在不与IANA或Internet注册机构协调的情况下完成此类操作。因此，地址空间可以被许多企业所使用。这一私有地址空间内的地址在企业内部或选择在这一空间进行合作的一系列企业内是唯一的，因而它们可以在其私有互联网上进行相互通信。"当然，这些地址都不能在外部世界使用，但这是IANA和互联网注册机构必须要确保的。

相比之下，从外部世界可见和可访问的任何实体都必须具有来自全球唯一地址空间的地址。这些地址只能由互联网注册机构（依次由IANA来分配地址空间）分配。

将从私有空间分配网络地址的主机称为私有主机；同样，将得到全球唯一地址的主机称为公有主机。图5.19对该配置进行了描述。

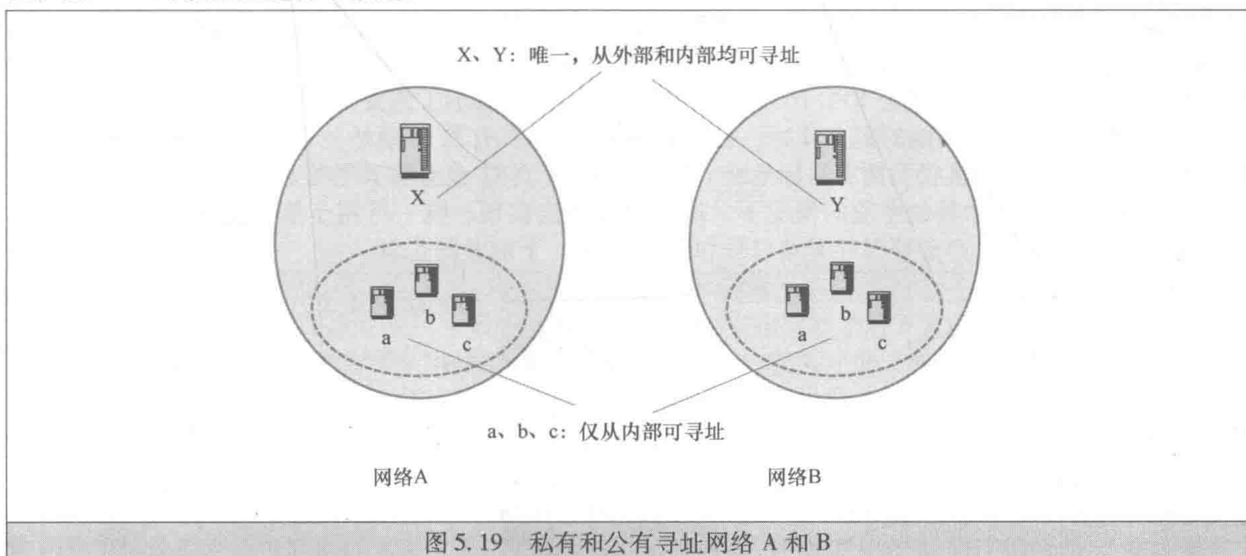


图5.19 私有和公有寻址网络A和B

位于两个网络A和B中的私有主机（配置在穿孔椭圆形内）拥有私有IP地址{a，b，c}，这些IP地址仅在主机所在的网络内有意义。需要注意的是，除了主机之间进行相互通信的能力之外，它们可以与独立于该主机地址的各自网络内的任何主机进行通信。具有全球唯一IP地址为X和Y的主机拥有公有地址。互联网上的任何主机都可以对其进行寻址。在图5.19的特定实例中，8台主机只使用了5个IP地址。

主机的IP地址可以由私有转变为公有（反之亦然）。

⊖ 1918.txt。需要注意的是，1918年是第一次世界大战结束的一年。Bob Moskowitz告诉我们一个引人入胜的开发故事。一切始于1983年，在休斯顿的IETF会议上，在Bob Moskowitz与Jon Postel的走廊对话中，Bob请求4个B类网络地址空间，因为克莱斯勒公司（当时Bob所在的公司）需要将这些地址用于工程、制造和其他机构——所有这些机构都需要单独的网络。Jon答复说，他只能提供两个B类地址空间，因为IP地址正在耗减。然后，Bob建议在这种情况下，有必要开发可在企业网络中重用的私有地址空间概念。不敢相信，Jon回答说，建议Bob编写一份RFC来详细阐述这一概念。此时，无意中听到这一谈话内容并聆听了一段时间的Yakov Rechter，加入谈话并表达了与Bob共同撰写该RFC的兴趣。这就是RFC 1597如何诞生的。然而，该RFC的出版却引起了争议，很快就形成了两大阵营：一个阵营支持开发私有地址空间，另一个阵营反对开发私有地址空间。Jon Postel要求两个营地和平协商，并在最佳实践RFC中记录他们达成的共识。在长时间的讨论之后，形成了一份协议，新形成的文档与RFC 1597的区别仅存在于描述私有地址空间的陷阱和危险的文本中。Jon Postel以冷幽默而闻名，然后1918分配给新RFC作为文档编号，声称1918前的所有其他数字都被使用。然而，与RFC 1918（1996年2月）同时发布的RFC有好几个，且在IETF中没有人对这一数字的含义提出质疑。

当然，不会将私有网络的路由信息传播到外部世界。边缘路由器（或更准确地说是防火墙）的一项职责是确保拥有私有源地址或私有目的地址的数据包：①不会离开它所发起的网络；②不会进入任何网络。至于职责②，人们不会将边缘路由器拒绝接收具备私有 IP 地址的输入分组或与该地址相关的路由信息等视为协议错误。

同样，DNS 资源记录或与网络内部私有地址相关的其他任何信息绝不能离开网络。这是由防火墙执行的另一种策略。

此外，利用私有地址的所有优势来维护 IPv4 空间，这种方法的主要问题是要将 IP 地址设想为全局的。引入私有地址的事实表明，它偏离了最初的互联网愿景，且禁忌被打破。

5.3.2 NAT 盒的架构与操作 ★★★

正是因为许多 IETF 工程师无法在破除禁忌（这一禁忌是指行业在标准确定之前的野蛮发展）问题上达成共识。虽然说 IETF 没有从事 NAT 标准化工作不够准确，相反，几个工作组过去在从事且现在一直都在从事 NAT 标准化工作。现实情况是虽然 IETF 没有制订 NAT 标准，但是它制订了如何与 NAT 盒共存的标准（将在下一节中进行描述）。

这里的微妙问题是架构本身不一定要进行标准化，但需要对协议进行标准化。准则是标准不需要涉及如何构建一个盒子，它只是用于描述盒子的行为。虽然该准则是不正确的，但 NAT 盒就像是出现在海报上的儿童，用于说明内部结构和行为是不可分割的。

虽然 1999 年发布的信息类 RFC 1631 绝对不是标准，但它揭示了需要实施的内容。

这里从 NAT 盒最小功能的定义开始。它必须将 S（一组私有源 IP 地址）转换为唯一的源 IP 地址 i，但这样做的目的是，在接收到拥有目标地址 i 的数据包时，NAT 盒会将其转换回唯一的 $i' \in S$。显而易见，在不修改 IP 报头中其他字段的情况下，这一目标无法实现。唯一可用于修改的字段是源端口号。但如果源端口号更改，则必须保存源端口号和源 IP 地址。下面看图 5.20。

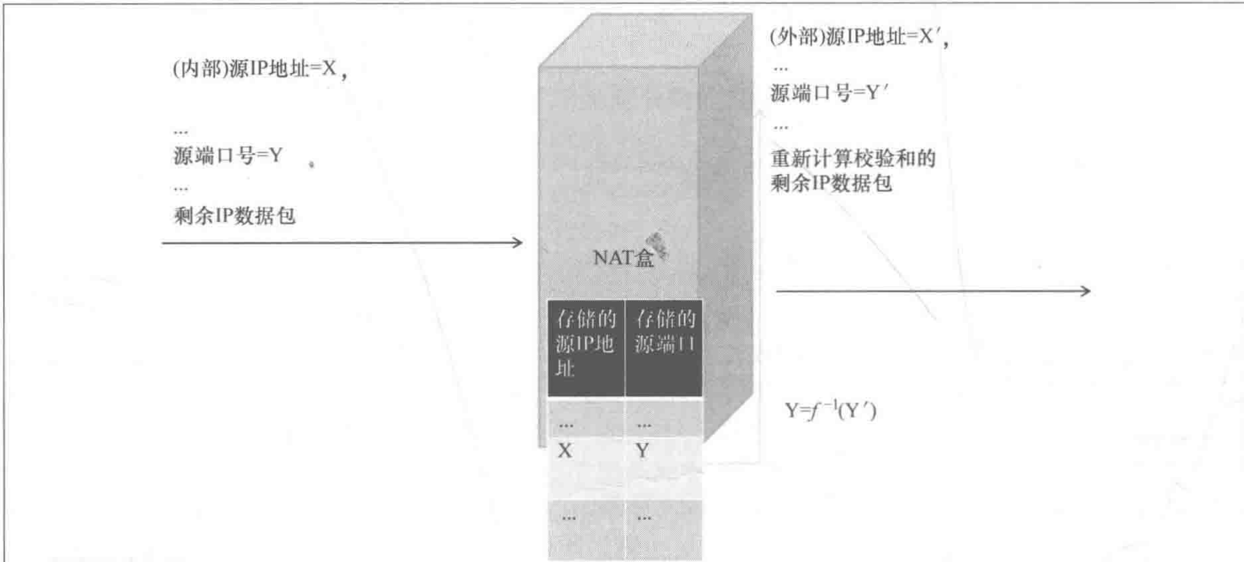


图 5.20 NAT 盒输出流量

假定 X 和 Y 是从网络内部到达 NAT 盒的 IP 数据包的私有源 IP 地址和源端口号。为了确保未来成果能够后向兼容，NAT 盒必须将（X，Y）存储在转换表中，并将指针转换为对于 Y′的表格项[㊀]。离开 NAT 的数据包现在拥有 NAT 盒的公有 IP 地址 X′以及源端口号 Y′。但是既然现在 IP 报头已经改变了，那么每个 IP 报头以及传输协议头的校验和都必须进行更改。提醒读者，源端口号是传输层报头的一部分。在这一点上，另一个禁忌被打破——违反了分层原则，因为网络层实体需要查看和修改传输层 PDU。本质上，NAT 盒执行的是路由器功能。

㊀ 事实上，Y′可以只是那个指针（数组索引 i），但端口号选择的规则因端口号随机化的要求而变得更加复杂，这表明通常情况下可以使用可逆函数 f，使得 $i=f(Y)'$。因此，反方向的转换是比较简单的。

不仅如此，而且几个互联网应用层协议也已经违反了分层原则，因为在采用开发和实现时间都晚于这些协议的通用资源定位符方案的情形中，这些协议已经承载了应用协议有效载荷中的 IP 地址。为此，RFC 1631 建议“NAT 还必须关注 ICMP 和 FTP，并修改 IP 地址出现的位置。毋庸置疑，还有许多其他位置需要进行修改。”这是一项相当复杂的任务[1]，因为传统做法是需要将应用协议（如电子邮件协议、多媒体会话启动协议和超文本传输协议）指定为 ASCII 文本。这就是打破一个禁忌如何立即导致其他诸多禁忌被打破的情形……

就输入流量而言，NAT 盒的行为是唯一确定的，如图 5.21 所示。

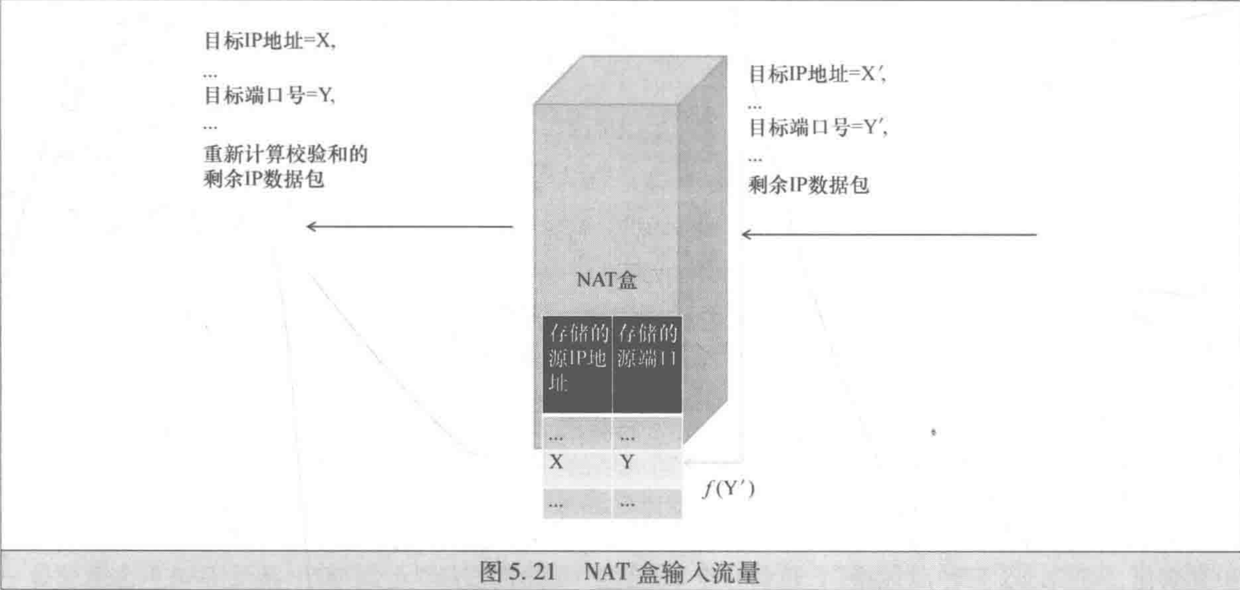


图 5.21 NAT 盒输入流量

由于 X′是一个与 NAT 盒相关的常数，因而只能使用 Y′来确定原始对（X，Y）。

在这一点上，应该清楚为什么采用这种方案。在实体发起联系之前，NAT 盒后面的实体不能与外部世界有任何联系。事实上，正如预期的那样，这一问题变得更加复杂。刚刚提出的转换方案要求用于保存（X，Y）对的表大小与私有网络中的主机数成正比。对于大型运营商来说，此表实际上太大而无法实施，因而表中条目数必须小于逻辑最大值。为了满足这一要求，条目必须在占用一段时间（时长由时间参数决定）后从表中删除。这里，实现方案有所不同，但无论如何实现，NAT 盒后面的私有网络中实体外部对话可能永远无法达成，直至重新启动对话。这需要用于增加网络流量“保持激活”例程，而不会带来任何有效载荷。

由于不断引入新功能，导致 NAT 变得越来越复杂，进而使得这一问题变得越来越糟糕。重要的是，需要注意转换机制对于功能引入来说是一成不变的。变化的是存储什么内容和存储多长时间。

这些功能（从未实现标准化，但对 NAT 盒的运行状态影响极大）太多而不便于讨论，且与功能有关的分类方法多种多样。为了编写本书，只是暗示这里发生了什么。

考虑私有网络中的主机只接收来自外部主机的消息时的要求，主机使用这些消息来发起对话。这一要求不仅是合理的，而且为了防止上一节讨论的反射拒绝服务攻击，也必须实施这些方案。

考虑图 5.22 中的情况。假设私有主机 X 与外部主机 A 和 B 开始对话。当然，外部主机 A 和 B 可以进行响应，因为 A 和 B 知道支持它们识别私有主机 X 的唯一端口号。但是无法阻止（潜在恶意）主机 C 得到这个端口号（如通过嗅探主机 B 的流量或简单地让 B 泄露这一数字——这绝对不是秘密，且不受密码保护）。一旦主机 C 知道这个端口号，它就开始向主机 X 发送消息。根据要求，NAT 盒不允许该消息通过，但是根据刚才描述的配置，NAT 盒绝对没有办法这样做。

对于 NAT 盒子来说，唯一的解决方案是存储每个打开会话的参数——不仅包括源 IP 地址和源端口号，还包括目的 IP 地址和目的端口号。这会立即成倍增加表格的大小。

[1] 当应用有效负载被加密时，这一任务无法完成。

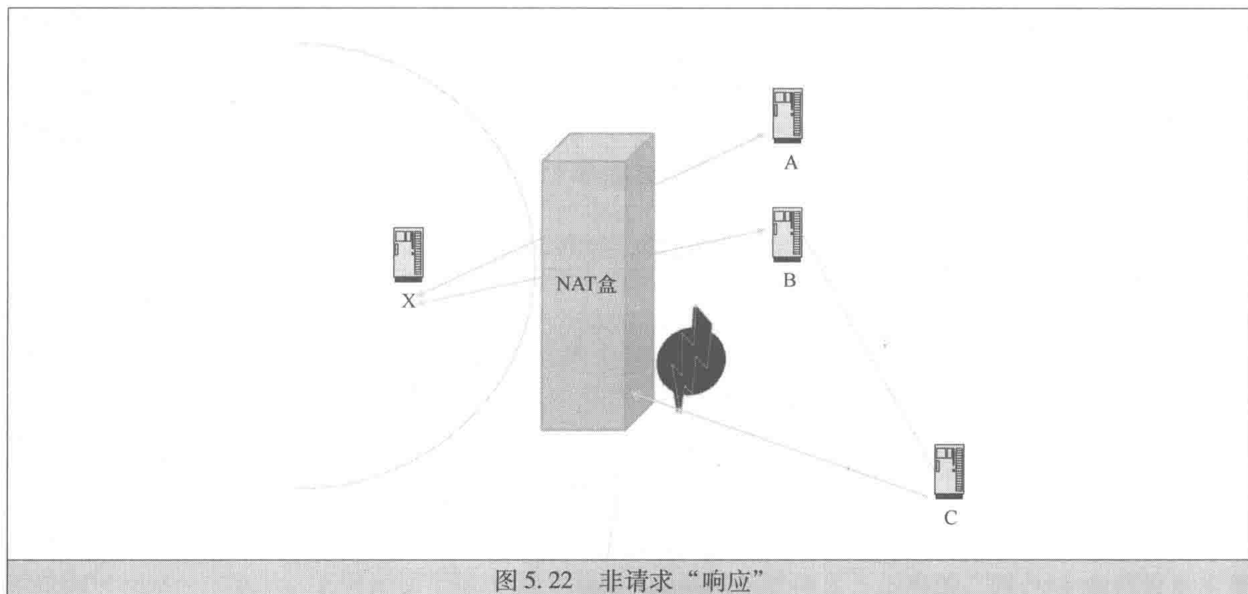


图 5.22　非请求“响应”

更严格的还会要求保存协议号、每次会话的定时器间隔和其他参数。遗憾的是，这一切还只是冰山一角……

读者可能已经注意到，如果考虑将 NAT 盒子与防火墙分开，则其中的许多问题可能会迎刃而解。然而，这些实现方案始终坚持将 NAT 功能作为防火墙的一部分，因而进一步模糊了本质上不同的功能实体之间的区别。

为了应对复杂性，IETF 成立了 NAT 工作组，但是它并没有制订急需的标准。RFC 2663 使事情进一步复杂化，它引入了多种“风格”，包括一个 NAT 盒（或两个 NAT 盒），其中涉及 DNS。这绝对是一份有趣的阅读材料，但正如其他与 NAT 相关的 RFC 文档所提及的，RFC 2663 属于信息类——它未提供标准。

2001 年初出版的专著[13]，对 NAT 架构和部署进行了详尽的分析，对与 NAT 交互的应用协议进行了回顾，但相当多的成果（特别是将在下一节讨论的）是在后期开发的（开发动力首先源于 IP 电话，以及后来更为通用的实时多媒体需求）。

同时，IETF 还成立了用于避障的行为工程（Behavior Engineering for Hindrance Avoidance，BEHAVE）工作组，旨在创建“支持 IPv4/IPv4 和 IPv6/IPv4 NAT 以尽可能确定的方式发挥作用的文档”。该工作组已经完成历史使命，共编写了约 30 份文档，足以填满一个小图书馆，从而使得专门研究 NAT 行为的文献量接近专门研究人类精神病学的文献量……

关于 NAT 的发展简史，以及存在细微差别的赞成与反对论据，我们强烈推荐大家阅读由 IETF 内部人员、RSVP（Resource Reservation Protocol，资源预留协议）发明人张丽霞撰写的参考文献[14]。以下是张博士的观点之一：“对 NAT 的误判让我们付出极大代价。在开展大辩论的同时，NAT 部署逐步展开，标准的缺失导致了各种 NAT 产品间出现大量的不同行为。在此期间，人们还开发出诸多新的互联网协议……他们所有的设计均基于 IP 架构的原始模型，其中 IP 地址被假设为全球唯一且可访问的。当这些协议准备用于部署时，它们面对的是一个与其设计不匹配的世界。这些协议不仅要解决 NAT 穿透问题，而且还要提供用于处理各种 NAT 盒行为的解决方案。”

“行为”一词在这里是关键，因为只是弄清 NAT 盒下的映射而无法保证 NAT 穿透。我们已经非正式地描述了行为之间的差异。在 RFC 3489（现已作废）中，作者采用了早期（2003 年）尝试用于表征此类行为的成果：

1）使用完全圆锥形 NAT，外部主机可以在无先决条件的情况下向内部主机发送数据包。

2）使用受限圆锥形 NAT，外部主机无法发起与内部主机的对话。只有当它被告知特许时，它才能说话。换句话说，NAT 盒跟踪内部主机先前发送消息的目标 IP 地址，并支持仅来自那些地址的流量。

3）使用端口受限圆锥形 NAT，先前的限制条件被限定为仅支持通过对话发起端口进行响应。这里，NAT 盒跟踪内部主机先前发送消息的数对（目的 IP 地址，目的端口号），并支持与上述数对匹配



的（源 IP 地址，源端口号）IP 数据包。

4）使用对称型 NAT，表项是会话（源 IP 地址，源端口号，目的 IP 地址，目的端口号），且每次会话均被映射到不同的端口号上。

2007 年，针对单播 UDP 数据包，IETF 发布了关于 NAT 行为的最佳实践 RFC 4787。RFC 承认："在某些情况下，同一 NAT 将表现出不同行为，这一事实使得 NAT 的分类变得更加复杂。这已经在用于保护端口或拥有用于选择端口而不是空闲端口特定算法的 NAT 上得到印证。如果 NAT 希望使用的外部端口已经被另一个会话使用，则 NAT 必须选择不同的端口。这会导致针对这种冲突情况的不同代码路径，进而导致不同行为的出现。"一个糟糕的结论是，IETF 不是为 NAT 和领先行业制定标准，而是对无标准状态束手无策。越来越大的文件出现，偶尔"作废"（原文如此）的早期文档，每份文档都接近 TCP 规范的长度（该规范是迄今为止互联网中最复杂的规范）。这些文档的可读性不高，部分是因为所理解的 NAT 早期尚未开发的状态与描述 NAT 的文档所写状态相称……

现在，讨论用于处理 NAT 穿越问题的方法。

5.3.3 与 NAT 共存 ★★★

1999 年发布的 RFC 2663 中描述的一种早期机制是相当简单的，且目前仍在使用。我们称其为应用级网关（Application Level Gateway，ALG）。RFC 2663 将 ALG 描述如下：

"应用级网关（ALG）是与应用有关的转换代理，它支持某个地址域中主机上的应用能够透明地连接到另一个不同域中在主机上运行的应用。ALG 可以与 NAT 进行交互以建立状态，使用 NAT 状态信息，修改与应用相关的有效载荷，并执行任何其他必要的操作，来使应用在不同地址域内运行。"

其实，这种想法非常简单。该机制与支持臭棋篓子（事实上，可能是一位甚至不知道如何下棋的人）同时与世界上两位最好的棋手下棋的情形是完全一样的（假设 X 执白，Y 执黑），最终赢得至少一个回合或两次和局。

为了达到这一目标，臭棋篓子将通过重复 X 的第一步来开始与 Y 的游戏，然后通过重复 Y 在游戏中针对 X 的走棋方法，依次类推。在现实中，所有这一切都意味着是 X 与 Y 在一起下棋。如果游戏和局，则这意味着臭棋篓子和了两局；如果游戏赢了，则这意味着臭棋篓子赢了一局（输了另一局）。中间人的力量是无穷的！

这正是 ALG 的工作原理，如图 5.23 所示。在主机 X 上运行的进程总能启动对话（因为它位于私有网络内），认为它正在与主机 Y 上的对话者进行会话。相反，ALG 通过维护两大进程——一个模拟从 Y 到 X 的进程，一个模拟从 X（配置有 NAT 公有地址）到 Y 的进程，将自己插入到会话中间。ALG 不仅能够复制数据，而且还可以执行防火墙功能，修改甚至审查数据包。为此，ALG 是一种状态防火墙。

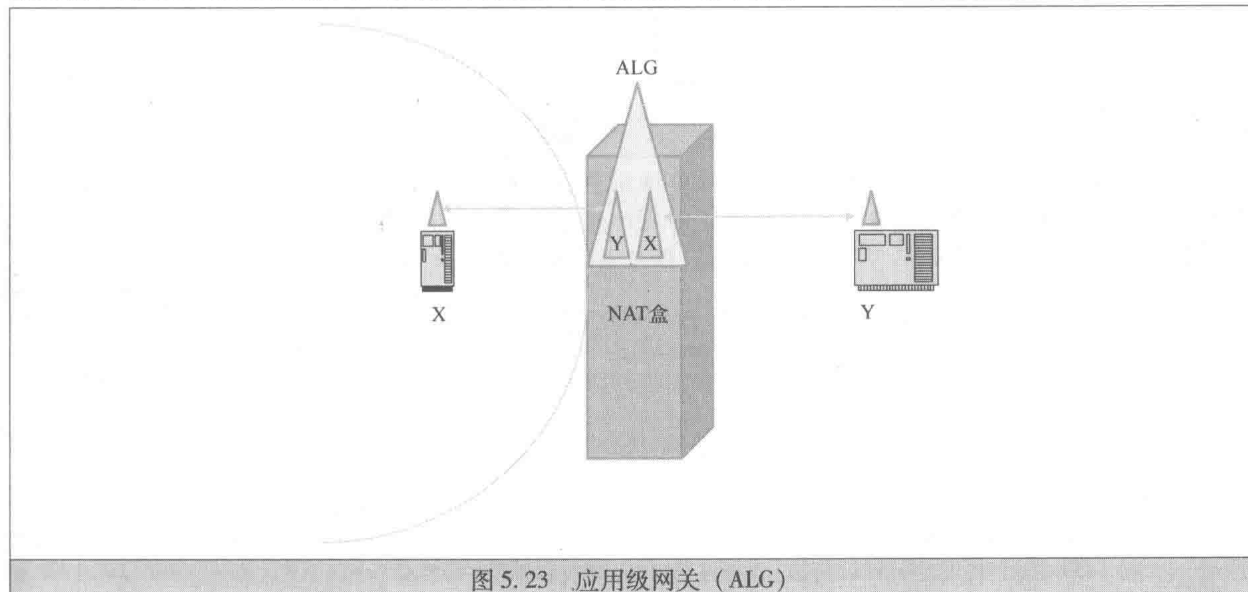


图 5.23 应用级网关（ALG）

在介绍 NAT 穿透的其他解决方案之前，应该强调一个事实，即只有 NAT 屏蔽网络中的主机才能与外部任何主机进行对话，这意味着位于不同 NAT 屏蔽网络中的两台主机之间无法进行通信。（再次看到突破终端原则导致了一连串影响深远的后果）。

巧合的是，这也是 IP 电话发展最初面临的一个主要问题（与本章参考文献［15］中描述的 PC 到 PC 的情形相同）。有趣的是，当前的问题仍然非常真实——2011 年，IETF 成立了旨在消除插件、支持在浏览器之间直接启用多媒体流的基于 Web 浏览器的实时通信（RTCWeb）工作组，该工作组在其特许项目——“针对防火墙和 NAT 穿透定义解决方案：协议和 API 要求”上耗费了大量精力。

从 IP 电话的发展开始，NAT 穿透的目标就是双重的：①发现 NAT 盒隐藏的实体；②为防火墙中的必要端口建立一个针孔[1]。发现过程是最难处理的问题。它拥有多种解决方案，其中的一些解决方案可能在某些情况下有效，但在其他情况下无效。

从最简单的解决方案开始，这些方案总是有效的。缺点是它成本过高，且可能在规模方面不切实际。我们将该解决方案称为使用中继 NAT 穿透（TURN），RFC 5766 对该方案进行了描述。TURN 最初设计用来作为 IP 电话的一部分，与会话发起协议（Session Initiation Protocol，SIP）和会话描述协议（Session Description Protocol，SDP）[2]协同工作，但是它的应用范围更广，稍后将进行讨论。

图 5.24 提出了一种想法：让中央（即公众可寻址的）服务器充当中继点，将两台位于不同网络中被 NAT 隐藏的主机（或对等体）连接起来。每个对等体都可以建立与中继服务器的连接，然后中继服务器将中继数据包，这样能够确保对等体进行相互通信。

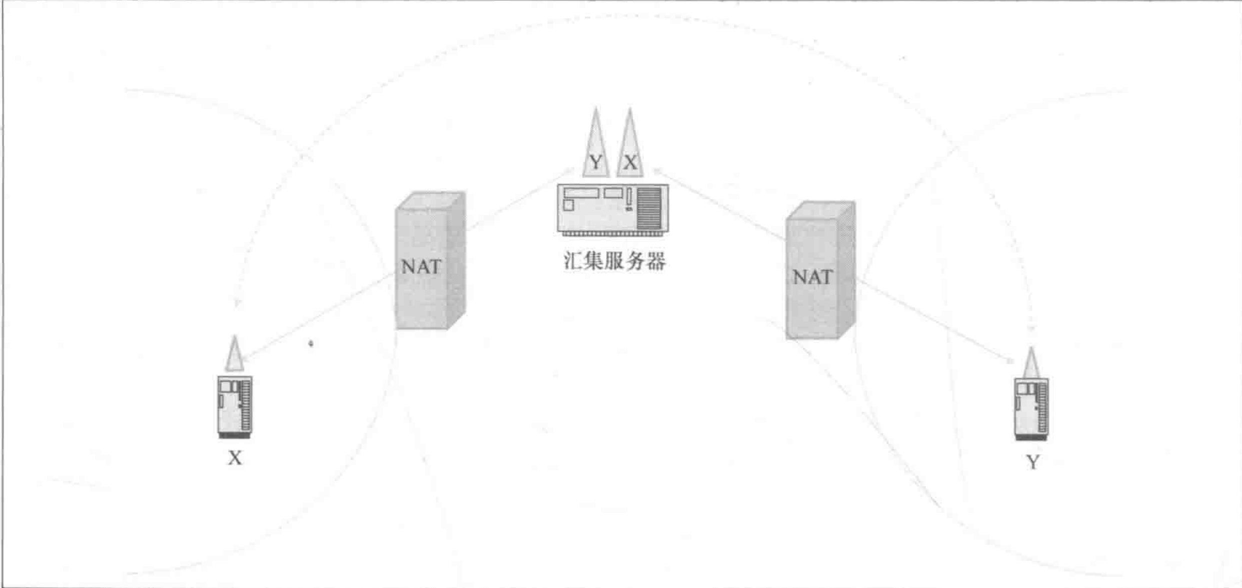


图 5.24　汇集中继

鉴于通信的本质是多媒体，因而可以推出中继服务器与网络之间必须具有非常高的带宽连接和非常大的容量。如果对等体可以在它们之间建立多媒体传输连接，能够使得中继仅用于初始信令（它需要的带宽和处理能力都不高），则这一要求可以被大大放宽。遗憾的是，这并不总是可能发生的。稍后

[1] 遗憾的是，针孔是一个常用但定义含糊的术语，因为 NAT 和防火墙概念交织在一起。为了编写本书，将其定义为 NAT 转换记录中内部主机的端口。通过启动对话，主机会创建一个针孔（主机可以通过该针孔返回）。这里的问题是防火墙功能可能会禁止某些端口的流量，因而 NAT 功能必须指示防火墙功能“打开针孔”。有时，该指令可能会对防火墙策略形成干扰。

[2] 会话发起协议（SIP）设计用于多媒体会话控制。SIP 现已广泛应用于企业（SIP 电话），它拥有重要的电信应用，最终实现了 3GPP IP 多媒体子系统（IP Multimedia Subsystem，IMS），其中 SIP 是核心协议。SDP 几乎不能算是协议，而是一种简单的 ASCII 文本文件，它列出了参与会话的进程属性。这些属性包括 IP 地址和端口号，它们后期被实时传输协议（Real Time Transport Protocol，RTP）和实时控制协议（Real Time Control Protocol，RTCP）应用于媒体会话中。本章参考文献［15］对这些协议进行了详细介绍，并描述了它们在建立和维护多媒体会话中的位置。还有许多专门针对 SIP 和 IMS 的专著。

会对这一点进行讨论。

现实生活中使用的是上述模型的变体。使用中继 NAT 穿透（TURN）是支持图 5.25 所示方案的协议名。

图 5.25 使用中继 NAT 穿透（TURN 协议）

重要的是，需要注意 TURN 服务器不是汇集点。其操作是不对称的（与电话交换机非常类似），因为 TURN 客户端必须知道它希望通信的对等体的 IP 地址和端口号。该信息可以通过对等体发现依托真正的汇聚服务器（如对等体注册的 SIP 服务器）来发现，或者可以通过电子邮件或类似带外方式进行分发。接着，需要 TURN 服务器的根本原因是只要它是公众的、已知的，它就拥有比其客户端更高的穿透防火墙概率。为此，仅当客户端无法能到达对等体时，它才求助于 TURN 服务器。

TURN 协议仅在客户端和 TURN 服务器之间使用，且必须由客户端启动。客户端可以从配置或通过 DNS 得到 TURN 服务的 IP 地址和端口号（SRV 参数的值为 TURN），也可以通过任意播地址来访问 TURN 服务器。

客户端使用 TURN 命令来创建和控制 TURN 上的数据结构（称之为分配），TURN 服务器是客户端希望进行通信的所有对等体的切换点。许多互联网协议的情形就是这样，TURN 服务器定义了一种保持激活机制：只要客户端在指定时间段内重复刷新请求，则需要保存分配。客户端还指定对等连接的权限。现在，TURN 服务器不与任何对等体建立连接——所有与对等体的通信都是采用 UDP 完成的。

客户端将应用数据封装在 TURN 消息中。TURN 服务器提取这些数据，并通过 UDP 进行发送。当对等体向另一个方向发送数据时，TURN 服务器把数据包含在 TURN 消息中，并将其中继到客户端。

寻址问题可通过分配来解决，该分配包含客户端的传输地址（即［IP 地址，端口］对）。这是指向对等体的所有消息中使用的源地址，因而成为来自对等体消息的目的地址。TURN 消息总是包含已经是消息源的对等体指示。

回到原来的问题：使用硬编码的内嵌 IP 地址和端口号的应用协议（如 SIP 或 SDP）在 NAT 方面能做些什么？NAT 能改变协议头，而无法改变内嵌数据？除非 NAT 发现其外部 IP 地址和端口号码，否则它用处不大。

顺便提一句，随着 NAT 相关出版物的大量出现，术语似乎存在某些不一致之处。有些规范将外部传输地址的组合称为映射地址，还有一些规范将其称为反射地址。在本节的剩余部分，将使用后一个术语，因为这是设计用于解决手头问题的协议中使用的术语。该协议的名称是 NAT 会话穿透效用（STUN）。

2003 年，STUN 协议第 1 版在 RFC 3489 中发布。那时，S 实际上代表“简单”，协议的名称是通过网络地址转换器（Network Address Translator，NAT）实现的用户数据报协议（UDP）的简单穿越。可以将 STUN 描述为“一种轻量级协议，它用于支持应用发现 NAT 及其与公共互联网之间的防火墙的存在和类型。”碰巧事情并不那么复杂（或者至少与原始互联网设计中表现出来的一样简单直接）。5 年后，新的 RFC（RFC 5389）发布。它已经被“淘汰”（原文如此[⊖]）RFC 3489，那时人们称之为“经典 STUN”。发生巨变的原因是“从 RFC 3489 出版以来的经验发现，经典 STUN 根本无法很好地工作，无法成为可部署的解决方案。”为此，“经典 STUN 无法提供用于发现它实际上能否正常工作的方式，且在无法正常发挥作用的场合不能提供补救措施。同时，人们发现针对 NAT 类型分类的经典 STUN 算法存在缺陷，因为许多 NAT 不能完全符合所定义的类型。”还存在着一种安全漏洞（在某些情况下，映射地址还可能会被欺骗）。

我们认为，在这种经验之后，IETF 开始阻止在其协议名称中使用“简单”和“轻量级”等词。当然，不存在任何关于 NAT 的简单介绍。

然而，STUN 的思路非常简单，如图 5.26 所示：要获得其反射地址，在主机上运行的应用进程应当只向 STUN 服务器发送一条请求，然后 STUN 服务器使用有效负载中的反射地址进行响应。的确，它像看镜子一样简单！

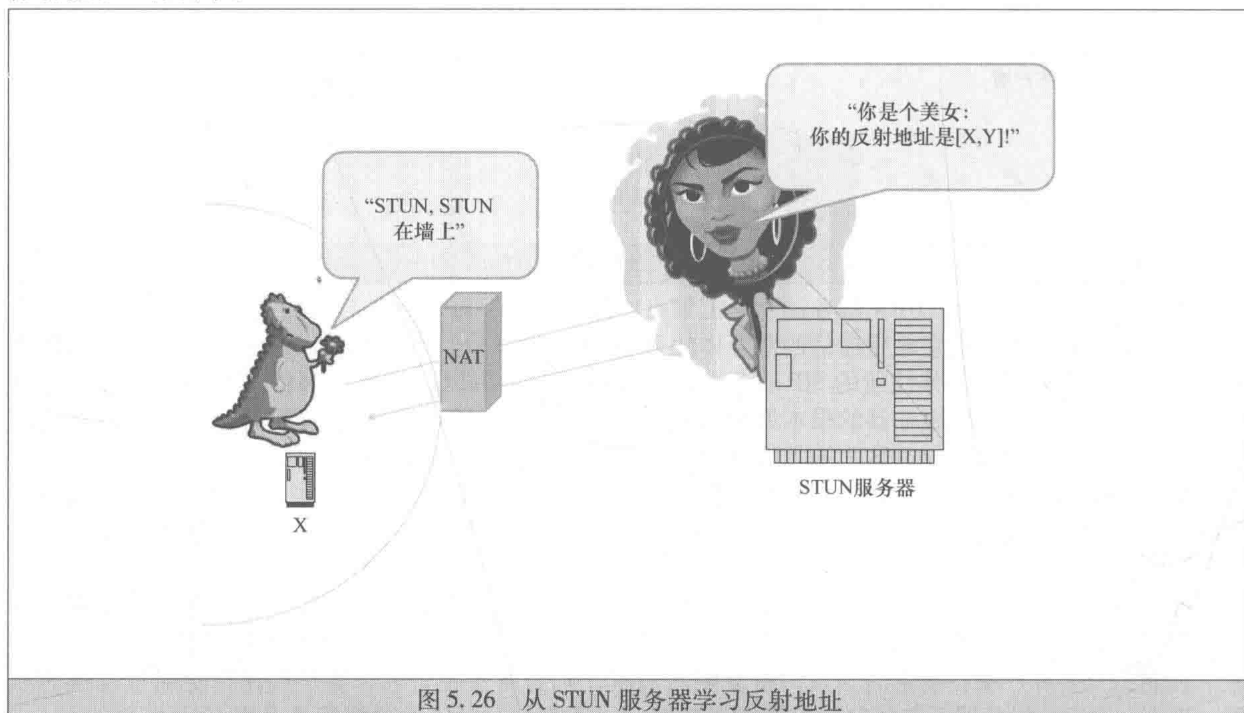


图 5.26　从 STUN 服务器学习反射地址

导致 STUN 初始设计复杂并最终破坏 STUN 初始设计的是 NAT 盒拒绝根据预测的逻辑采取行动。有些人当即认为互联网泡沫已破，但其他人出于现实政治需要，仍在极力推崇互联网。

RFC 3489 选择放弃提供完整解决方案的想法，而是将 STUN 视为一种包含多种已定义“用法”（即可应用 STUN 的特定情况）的工具。此外，STUN 协议目前可在 TCP 和 UDP 以及运行在 TCP 之上的传输层安全（Transport Layer Security，TLS）协议上使用。

该协议支持请求/响应事务和指示（该指示不需要响应）。同时，指定的单一方法是绑定。在事务

⊖ “淘汰”是 RFC 系列规范的术语。根据传统，当对某份 RFC 进行修改后，人们不再继续使用相同的 RFC 编号。如果修改只是一种更新，则旧 RFC 会被标记为“更新”，但是当旧 RFC 不再有效时，它会被标记为被新 RFC“淘汰”。

中，客户端使用绑定来查找它所绑定的反射地址。例如，指示可以用于使绑定保持激活状态。

STUN 的重要特征之一是其消息可以与其他协议的消息进行复用。为了确保解复用的正常进行，需要特别生成事务关联 ID 值（称为魔法饼干），这是用于避免歧义的一个字符串。（存在着一种扩展方案，它在同一任务中使用指纹属性来协助完成任务。）

STUN 的一个有趣的功能与功能强大的 NAT 盒有关，NAT 盒会尝试检测内置传输地址，并自行进行地址重写。因此，需要一种代码混淆技术：STUN 服务器对反射地址的值和魔饼的一部分应用异或（Exclusive OR，XOR）位运算[㊀]，运算结果最终作为反射地址值进行发送。

STUN 通过提供认证和完整性检查机制来解决安全问题。这些机制是可选的——它取决于“用法”选择。它们基于两种凭证机制：使用预置用户名和密码的长期凭证机制；使用通过带外交换来共享的短期凭证机制。采用任何一种机制，均可以通过挑战/密码响应来实现认证，并从密码导出用于完整性检查的密钥。

正如前面描述的其他请求/响应协议一样，STUN 可用于放大分布式拒绝服务攻击，但攻击的性质与之前所看到的不同。攻击者不是发送（概率相当低）伪造的响应和为多个客户端提供其反射地址，而是需要为多个客户端提供攻击目标。一旦客户端（希望在这个地址上收到其巨大的视频流）将地址提交给其对等体，则所有流量将被指向受害者。RFC 5389 指出，为了实施攻击，攻击者需要将自己插入到 STUN 服务器和多个客户端之间。

如果攻击者（再次充当 STUN 服务器和客户端之间的中间人）为客户端提供伪造的传输地址，则可以执行更简单的 DoS 攻击来“压制”客户端。这种攻击并不是 STUN 特有的，因为潜在攻击者也可以拒绝为客户端提供其他服务。攻击者可以通过向客户端提供自己的传输地址来轻松地修改攻击的性质，从而可以窃取流向客户端的流量。

更为严重的攻击是攻击者假扮客户端的身份。然而，这是一个比 STUN 更为常见的问题，且其解决方案存在于分发和保护共享机密的手段之中。

迄今为止，还没有讨论如何发现 STUN 服务器的问题，这是讨论的最后一个与 STUN 相关的问题。需要注意的是，当 STUN 进行复用时，服务器发现存在着问题，RFC 5489 定义了一种可选的 DNS 过程：“当客户端希望对接受绑定请求/响应事务的公众互联网中的 STUN 服务器进行定位时，SRV 服务名称为 stun。当客户端希望对通过 TLS 会话接受绑定请求/响应事务的 STUN 服务器进行定位时，SRV 服务名称为 stuns。STUN 用法可能定义了其他 DNS SRV 服务名称。”

TURN 和 STUN 服务器的实现存在于开源中，这使得它们在云中部署的场景中特别有用。

作为本节最后一个主题，来探讨一下使用 NAT 盒所带来的新复杂性。如前所述，NAT 盒的大小不能超过 61440[㊁]。物理内存限制是另一个约束条件，这可以使条目数量更少。处理此问题的一种方法是在网络边界部署不同的 NAT 盒，以使目标 IP 地址的不同前缀使用不同的 NAT[㊂]。图 5.27 充分反映了这种情况。

因此，给定网络内主机上的进程可能具有不同的反射地址，该地址取决于它从哪台 TURN 服务器获得地址。

现在，准备描述一种使用 STUN 和 TURN 协议实现 NAT 穿透的决定性机制。这种机制的名称是交互式连接建立（Interactive Connectivity Establishment，ICE）。ICE 绝对不是通用的，因为它仅适用于一组所谓的提议/应答协议，这些协议支持两个进程使用 SDP 达成多媒体会话的共识。SIP 是这样一种协议，且它充当用于定义提议/应答族共同特征的原始灵感。为了编写本书，可以假设所讨论的协议确实是 SIP。

㊀ 异或运算的定义如下：XOR（0，0）=XOR（1，1）=0，XOR（1，0）=XOR（0，1）=1。对于任意两个字符串 A 和 B，XOR[XOR(A,B),B]=A，在给定 XOR（A，B）和 B 的情况下，这使得 A 值变得微不足道。该技术实际上形成了一次性密码算法。

㊁ 由于端口号（入口的密钥）的长度是 16 位，且前 4096 个端口是预留的，因而条目数的上限等于 $2^{16}-4096=61440$。

㊂ 通过使用术语 NAT 而不是 NAT 盒，这里强调，实际的转换过程是不同的。（两个不同的 NAT 盒可以执行相同的转换功能）

图 5.27　不同路径的不同 NAT

ICE 是在 RFC 5245[1]中定义的。ICE 的最终目标是找到最适合在两个对等体之间建立 RTP（Real Time Transport Protocol，实时传输协议）/ RTCP（Real Time Control Protocol，实时控制协议）会话的两个传输地址。因此，假设对等体之间采用 SIP 进行通信，SIP 知道如何穿越 NAT。（当然，用于该信令的传输地址可能不同于那些最适合采用 UDP 来承载媒体流的传输地址）。这些传输地址可以通过 SDP 进行交换。ICE 通过累积传输地址，测试它们的连接性能，并选择最优连接来实现这一目标。

总共存在着 3 种类型的地址，如图 5.28 所示。

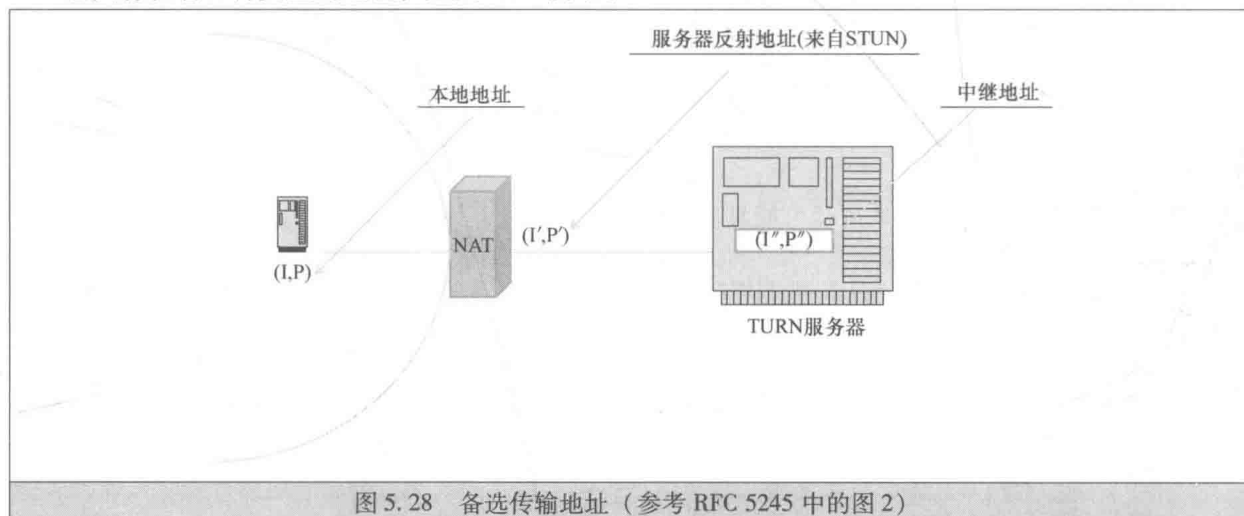


图 5.28　备选传输地址（参考 RFC 5245 中的图 2）

1）本地地址（即与主机的网络接口卡相关的传输地址）；

2）服务器反射地址（即从 NAT 盒另一侧的 STUN 服务器获取的地址）；

3）中继地址（从 TURN 服务器获取的、用于响应分配的地址）。

对于每个给定的对等体进程来说，在每种地址类型中，可能存在着多个备选地址。实际上，当有

[1] 该 RFC 发布于 2010 年，与其他 NAT 相关的 RFC 一样，它已经淘汰了 ICE 的两个较早版本。

多个网络接口（多宿主）时，存在着同样多的本地地址。如图 5.28 所示，在不同路径上可能存在着几个 NAT 盒[㊀]，且不同 STUN 服务器可能会提供不同的服务器反射地址。最后，可能存在着多台 TURN 服务器和多个中继地址。

ICE 操作可分为 3 步，如图 5.29 所示。

3) 测试连接性能，并共享结果

3′) 测试连接性能，并共享结果

a,b c,d …

a′,b′ c′,d′ …

1) 收集备选地址，对地址进行排列，并将其发送给对等体(通过SDP请求)

X

2) 收集备选地址，对地址进行排列，并发回列表(通过SDP响应)

Y

图 5.29 ICE 操作

首先，启动会话的客户端会收集备选地址，并根据计算优先级对其进行排序（请参阅包含用于计算此优先级的算法的 RFC）。一旦完成，这些结果将被发送给对等体（通过 SDP 请求）。对等体执行相同的操作，因而最终交换排序后的备选列表。

在这一点上，每个对等体都会检查每对传输地址的连接（每个对等体主机上运行的 STUN 均可以用于此目的），并将有关检查结果通知对话者。最后，两个对等体都有一个或多个工作对。

RFC 5245 仔细考虑了优化选项和安全性问题，尤其是连接性能测试。

5.3.4 运营商级 NAT ★★★

迄今为止，已经默认假设只在一层部署 NAT 盒。不过情况并非如此，因为互联网服务提供商也面临 IPv4 地址耗尽的窘境，特别是随着家庭网络的发展。

到 20 世纪 90 年代后期（根据本章参考文献［15］中描述的历史），那时单台家用计算机只是拨号连接到提供商网络中。在该点处，为计算机分配一个临时 IP 地址。现在，这种情况发生了巨大变化。每个家庭都可能拥有诸多设备，且所有设备都将一直在用。

这些设备的地址来自于私有地址空间，且家庭网关能够提供 NAT 服务。需要注意的是，它不仅涉及大型企业或住家，而且对于图书馆、酒店、酒吧和咖啡馆来说，提供上网服务也是日常惯例。

对于一些大型运营商网络，为每个家庭网关（或企业 CPE 网关）分配一个公有 IP 地址是不可能实现的。图 5.30 所示的解决方案是将私有空间地址分配给每个网关，并在公众互联网边界部署另一台 NAT 盒（这次使用的是公有 IP 地址）。

将此类 NAT 称为运营商级 NAT（CG NAT 或 CGN）。正如现实已经发生的那样，在一个以热爱行业术语而闻名的行业中，仅有一个响亮的名字还不够。表示相同对象的另一个名称是大规模 NAT（LS

㊀ 正如在下一小节要讨论的，NAT 盒可以进行嵌套。

NAT 或 LSN)。如果这还不够，则需要增加其他术语——这就是转换的本质。通用 NAT 提供 IPv4 到 IPv4 的转换，因而它被赋予了 NAT 44 的名称。由于 CG NAT 部署会影响 IPv4 到 IPv4 到 IPv4 的转换，因而它也被称为 NAT 444。因此，拥有 3 个名字和另外 4 个首字母缩略词——所有这些名称都适用于同一事物。这还没有结束：因为一些 NAT 盒子还可提供从 IPv4 到 IPv6 的转换，我们还得到了 NAT 46 的名字。作者们还没有看到 NAT 446 或 NAT 644，但是他们也希望这些名字出现，特别是因为 CG NAT 盒可以处理 IPv6 地址。没有人希望在整个世界转向 IPv6 时部署 NAT，因而预示着巨变的潜在 NAT 666 可能正好意味着互联网的结束。

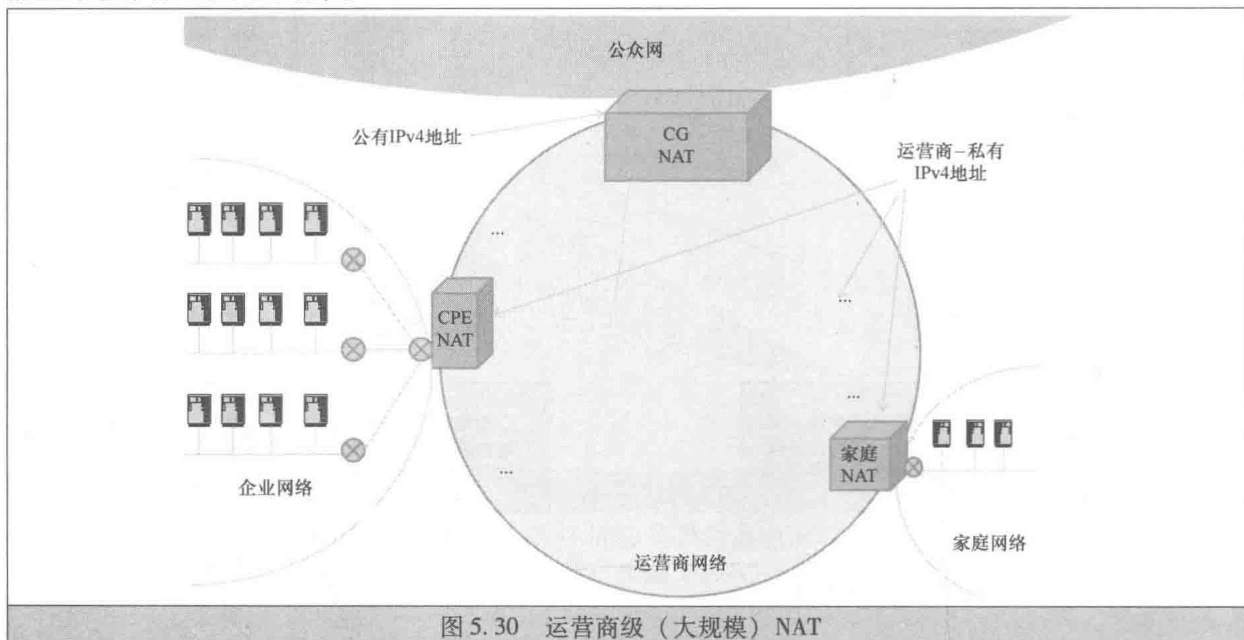


图 5.30　运营商级（大规模）NAT

实际上，RFC 6264 在承认“全球 IPv6 部署比原来预期的要慢”的同时，提出了“用于 IPv6 过渡的 CGN 增量方法（通过隧道）。在从 IPv4 向 IPv6 迁移的初始阶段，在大量传统 ISP 网络保持不变的同时，它可以为 IPv6 主机提供 IPv6 访问服务，为 IPv4 主机提供 IPv4 访问服务。”

这里，术语变得异常复杂。一件本质上全新且紧迫的事情是需要一个新的私有地址空间。正如公有 IPv4 地址可能不会用于 NAT 保护的网络中，因而家庭网络或企业网络中的主机所使用的私有地址可能不会被分配给下一级转换的网关和其他实体。

21 世纪头十年出现的一种令人震惊的做法是“地址抢占”。这仅仅意味着使用快速耗尽的 IPv4 地址空间中尚未注册的地址。正如 ICANN 的 Leo Vegoda 在本章参考文献［16］中所指出的：“许多机构选择在其内部网络中使用未注册的 IPv4 地址，在某些情况下，网络设备或软件提供商已选择在其产品或服务中使用未注册的 IPv4 地址。在许多情况下，做出使用这些地址的选择是因为网络运营商不希望承担从区域互联网注册管理机构（Regional Internet Registry，RIR）申请注册地址块的管理责任。”

最终，在 2012 年，互联网社区同意了这种最佳现行做法，正如 RFC 6598 中所描述的那样。这一做法是将私有地址空间扩展到 RFC 1918 中规定的私有地址空间，来为共享地址空间分配 IPv4 前缀。特别是针对当前地址抢占的做法，RFC 6598 禁止从“篡改的全球唯一地址空间（即地址抢占空间）”对 CG－NAT 二级接口（如网关接口）进行编号，并解释道：当“服务提供商将地址抢占空间的广告泄露到全球互联网中，该地址空间的合法所有人和那些希望与其通信的用户可能会受到不利影响”。甚至“如果服务提供商没有泄露地址抢占空间的广告，则服务提供商及其用户可能会失去与该地址空间合法持有人的连接。”

但是服务提供商接下来会做什么呢？下面两件事之一：

1）小心谨慎地重用私有地址空间，使同一地址永远不会分配给两个实体：一个在网关内部，另一个在网关外部；

2）在用户提供其 CPE 并对其内部网络进行编号的非托管服务中使用一些新的地址空间（顺便说一下，该地址空间是第一种方案不可执行的唯一情形）。

新的地址空间被特别指定为“达成促进运营商级 NAT（CGN）部署的目的”。随后，美国互联网号码注册管理机构（American Registry of Internet Numbers，ARIN）采用了如下策略：“需要预留一个二级连续的/10 IPv4 块来促进 IPv4 地址扩展。不会将该块分配或指派给任何单一机构，但可由服务提供商共享来实现 IPv4 地址扩展部署内部使用，直至连通网络完全支持 IPv6。这种需求实例包括家庭网关和 NAT444 转换器之间的 IPv4 地址。”

因此，IPv4 地址空间的耗尽再次被推迟……而且这还是在不受约束的情况下发生的。RFC 6598 列出了某些服务变得无法提供 CG NAT 的几个实例。

这些服务中的一些似乎是一类不需要转换即可实现的服务——与游戏机类似。当两个使用相同 IPv4 地址的用户尝试相互连接时，其他包括点对点对点应用和视频流。该类型中更重要的服务是需要确定 CG NAT 服务器位置的地理位置服务，以及需要全局可达地址的“6 到 4”（即 IPv6 到 IPv4）转换。

现在，很容易会继续兑现早期承诺，以澄清 DMZ 高深莫测的用例。在该用例中，企业配置了公众 Web 服务器。在图 5.17b 所示的两个防火墙中，NAT 盒是在将信任区与 DMZ 分开的防火墙中实现的。公众服务器可能无法在 NAT 后运行，因而 DMZ 是企业中公众服务器的唯一部署位置（仍由另一个防火墙负责保护）。类似地，在图 5.17a 所示的情形中，需要对 NAT 盒进行合理配置，以确保它不会干扰 DMZ 接口和公众接入接口之间的流量。

现在，可以转到通常在防火墙中实现的下一项功能——负载均衡。

5.4 负载均衡器

负载均衡的概念非常直观。这就是人们如何（从若干种相同的资源中）选择最合适的资源来执行给定功能。例如，菲律宾前第一夫人伊梅尔达·马科斯（Imelda Marcos）因收藏了大量的珍贵鞋款（约 3000 双）而闻名世界，其中的一部分被陈列在菲律宾国家博物馆。我们推测，她会综合考虑时间、天气和她打算穿衣的整体风格等因素来选择一双鞋子[⊖]。

类似的例子还有电话。我们选择电话，是因为它仍然在广泛使用，也是因为它非常透彻地说明了 IP 网络中负载均衡几乎所有的应用。这个例子就是所谓的 800（也称为 Freephone）服务，在本章参考文献［17］中有详细描述，如图 5.31 所示。

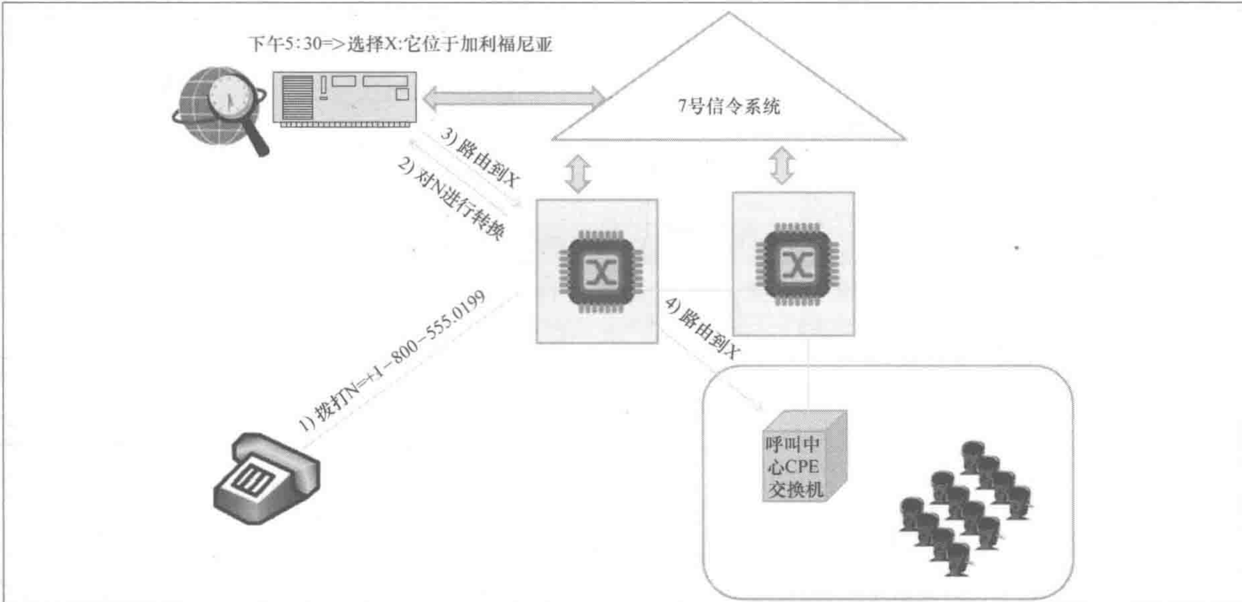


图 5.31 负载均衡实例：选择 800 服务的呼叫中心

当某人需要呼叫公司的客户服务时，到达该公司的号码表示一种呼叫者免费的特定连接。相反，

⊖ 需要注意的是，在这个例子中，“均衡”一词应用于马科斯夫人通过选择某双鞋来实现鞋子负载均衡的行为，这不应该与她在穿着鞋子时保持其平衡相混淆。

被叫方（在这种情况下，被叫方是公司）负责为服务付费。在美国，通常为这一号码指定特殊的前缀（800 前缀是 AT&T 在 20 世纪 80 年代初推出服务时出现的第一个前缀），用于表示它应该与“标准”数字区别对待。

当拨打这种号码时，电话交换机立即知道它无法路由到该号码，且需要特殊指令来对号码进行处理[⊖]。因此，交换机借助指令指向一种称为服务控制点（Service Control Point，SCP）的计算机。（交换机和网络控制点之间的所有信令数据流均采用 ITU - T 标准化的 7 号信令系统协议在单独数据通信网络上运行。）

反过来，服务控制点（SCP）调用了一种自定义编写的业务逻辑程序，它使转换成为可能。在实例中，程序会查看呼叫时间来决定应该到达哪个呼叫中心。由于时间恰好是下午 5:30，程序表明，新泽西州的呼叫中心将在当天关闭，因而最合适的呼叫中心是加利福尼亚州（那里当地时间是下午 2:30）。因此，需要指定到加利福尼亚内部呼叫中心交换机的路由。对于 3 小时后启动的呼叫，接下来 8 小时将指定班加罗尔的呼叫中心，之后是新泽西州的呼叫中心。太阳在公司永远不会落！需要注意的是，根据业务逻辑程序，转换是动态处理的。另外，需要注意的是，这一实例绝不是一种采用过时技术的实例——谷歌一直在提供这种服务的入门级变种，作为 Google Voice 的一部分。

在这一实例中，基于呼叫中心的可用性，可以将每次呼叫分配给某个呼叫中心。但是，通过智能网（IN），还可以基于负载分配，从多个呼叫中心中选择一个呼叫中心：呼叫将被路由到一个业务处理量比其他呼叫中心少的呼叫中心。为此，呼叫可以根据分配权重在呼叫中心之间进行统一分配。在极端情况下，服务控制点（SCP）可通过应用呼叫间隙来保护电话网络免于过载，而呼叫间隙是丢弃已定义呼叫的百分比（在这种情况下，呼叫者将听到一种短促“繁忙”信号）。

上述负载均衡功能在多个环境中几乎是一致的。本节的剩余部分描述了现代服务器农场中的负载均衡，提供了实现和部署负载均衡的实例，并讨论了如何将 DNS 应用于负载均衡。

5.4.1 服务器农场中的负载均衡 ★★★

如前所述，上述特征集与现代服务器农场中使用的功能几乎相同，如图 5.32 所示。服务器负责执行任务以响应客户端的进程（可能在专用主机上运行的每个进程）。万维网服务器（返回到网页）可能是第一个广泛使用的实例，其次是 SIP 服务器（针对 IP 电话）、DNS 服务器等。负载均衡器获取请求，然后将其转发到所选的服务器。

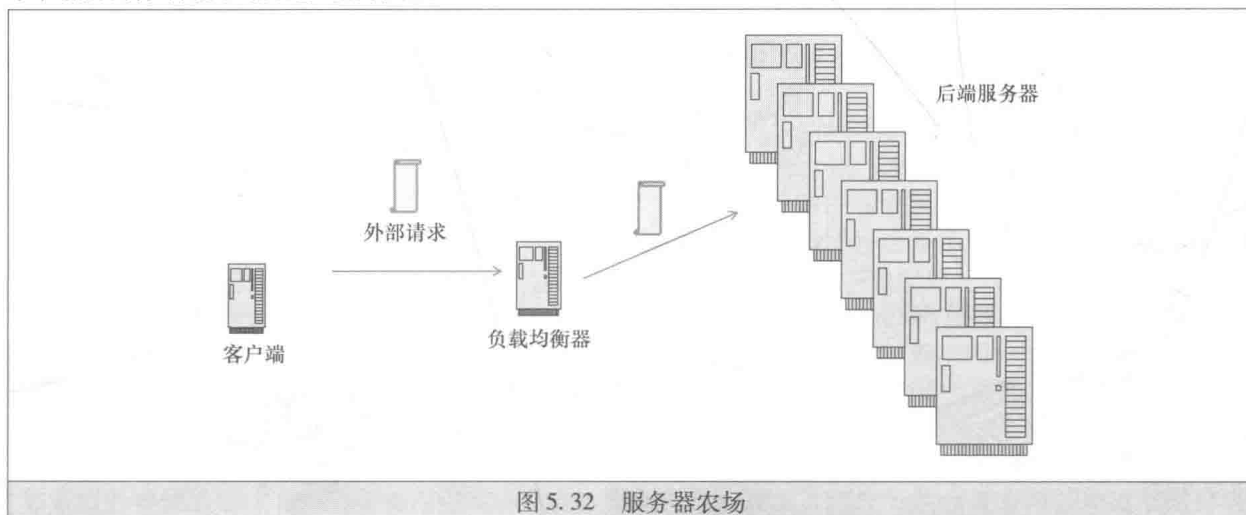


图 5.32 服务器农场

后端服务器不一定必须位于同一位置。事实上，它们可能在地理上分散（当然，在这种的情况下，不会将其集合称为“农场”）。实际上，备份服务器甚至不一定是“服务器”。负载均衡器可以在网络

⊖ 正如前面所提到的，这与 SDN 概念非常相似。事实上，将智能网（Intelligent Network，IN）支持的一种服务称为软件定义网络（Software Defined Network，SDN）。但是其转换部分也与 DNS 的功能非常相似：至少可以将不可路由的号码转换为可路由的号码，且可动态转换号码。

边缘的路由器中实现。在这种情况下，负载均衡器可以通过备用链路来分配流量。

这种配置的几个主要优势如下：

1）加速处理（使用并行技术——所有服务器都能执行每项指定的任务）；

2）提高可靠性（如果一台或多台服务器宕机，则服务器性能恶化，且在服务器恢复时性能提升）；

3）支持可扩展性（使用高效并行的任务、添加新服务器来线性提升性能）。

最后两项是云计算的重要特征，这使得云计算中负载均衡器的用途显而易见。

与 NAT 提供的安全性有些相似，安全性拥有一大优点，因为后端操作的内部结构对外隐藏。当负载均衡器在防火墙内实现（这是常见情况）时，它实际上可以终止安全会话，从而保护后端服务器免受维护安全状态的影响。更为重要的是，负载均衡器用于缓解各种拒绝服务攻击，特别是 SYN 攻击（通过实现 SYN Cookie 和延迟绑定）。

正如已经看到的，负载均衡器的性质是双重的：它既是中间人又是调度器。调度器是负载均衡器最重要的功能，因而在本节剩余部分将对调度器进行集中讨论。

这里，第一个相关问题是选择一种调度算法。存在着后端服务器可以随机选择的服务——使用与分配给每个后端服务器的任务比例有关的统计规范。后端服务器的地理（或网络特定）位置可能是另一个因素，这对某些应用来说可能是至关重要的。因此，调度算法也可以基于反馈机制：负载均衡器可以监视每台服务器的负载和运行状况，并相应地调整任务分配。

迄今为止，已经默认假设适用于负载均衡应用的任务类型是一次性请求/响应事务。然而，负载均衡可以与面向会话的应用协同工作。这里的主要问题是确定保存会话状态的位置，以便新分配的后端服务器能够知道到达请求的准确情境。

一些应用协议（特别是 HTTP）在客户端存储会话的整个状态。有趣的是，这种机制开发的初衷显然不是解决负载均衡问题的，而是解决可扩展性问题的。当客户端数量急剧增加时，将每次会话服务器端状态存储到服务器上是不现实的甚至是不可能的。相反，该协议会强迫客户端存储该状态，然后在下一个消息中将其呈现给服务器[㊀]，如图 5.33 所示。



图 5.33　在客户端保存会话状态（一个 Cookie）

将状态信息的主体称为 Cookie。这是一种支持诸多 Web 应用的核心机制。考虑网上购物——从一个网页移到另一个网页，把东西放在你的购物车里等。服务器不保存浏览的历史记录。相反，它会向客户端返回一个 Cookie，后者预计会在下一条请求中进行显示。

一旦服务器收到 Cookie，则它恢复状态。顺便说一下，这就解释了为什么应当对 Cookie 进行密码保护（以免恶意客户端通过修改 Cookie 来欺骗服务器，在实际没付款的情况下来标识已经完成付款）和加密（为了保护客户隐私和网络企业的商业利益）。

然而，存在着一些不采用 Cookie 机制的应用。在这种情况下，为了支持负载均衡，将会话状态存储在中央数据库中。当然，也会因单点故障而出现了严重的可靠性问题。

对负载均衡器的研究正在进行，特别是它们在确保网络自愈性能方面的应用。存在着一种有趣的方法基于与化学的类比（本章参考文献［18］对该方法进行了说明）。本章参考文献［19］是一本与服务器负载均衡有关的综合性专著。

㊀ 需要注意的是，这与 CPU 操作惊人地相似。每个进程的状态都保存在该进程的内存中（而不是保存在 CPU 中），因而在每次需要执行该进程时，CPU 都会获得该进程的状态。

正如下一节将要证明的，负载均衡并不神秘。

5.4.2 实例：负载均衡 Web 服务 ★★★

在实践中，使用开源软件在云中创建和部署负载均衡非常简单[⊖]。在下面的实例中，我们使用万维网服务（更具体地说，使用 Nginx 服务器软件）。建议读者将该实例作为一项实际任务，使用诸如亚马逊的 EC2（Elastic Compute Cloud，弹性计算云）服务来实现。

我们的任务是创建一个由 4 台相同服务器维护的网站，且负载均衡是由第 5 台服务器负责实现的，如图 5.34 所示。为了达到这一目标，可以创建 5 台虚拟机，这可以通过将运行 Linux 操作系统作为亚马逊 EC2 实例来实现，并在每台虚拟机上部署 Nginx 服务器。一台虚拟机将使用 Nginx 服务器来充当另 4 台虚拟机的负载均衡器。

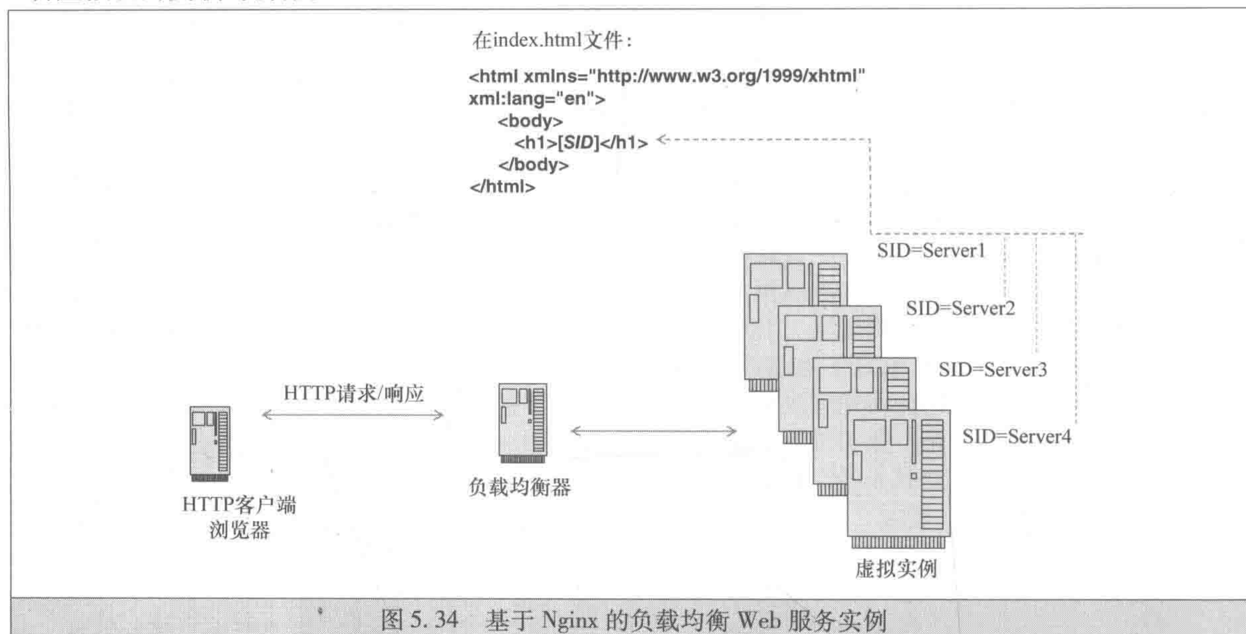


图 5.34 基于 Nginx 的负载均衡 Web 服务实例

对于 4 台服务器中的每一台，均需要借助用于确定服务器标识的文本对 index. html 文件（位于 directory/usr/share/nginx/html 中）进行更新，如图 5.34 所示（需要使用被 Server1、Server2、Server3 或 Server4 替代的字符串 SID）。

剩下唯一要做的事情（也是最有趣的事情）是配置负载均衡器。事实证明，这与配置服务器一样简单。负载均衡器的配置文件位于 etc/nginx/nginx. conf 中。图 5.35 包含了示例代码。

```
events {
        worker_connections 768;
    }
http {
    upstream myapp {
        #ip_hash;
        server [SERVER1_PUBLIC_DNS_NAME] weight=1;
        server [SERVER2_PUBLIC_DNS_NAME] weight=1;
        server [SERVER3_PUBLIC_DNS_NAME] weight=1;
        server [SERVER4_PUBLIC_DNS_NAME] weight=1;
    }
    server
    {
        listen 80;
        server_name myapp.com;
        location /
          {
            proxy_pass http://myapp;
          }
    }
}
```

图 5.35 配置负载均衡器

在实例中，每个服务器的权重值为 1，这意味着在运行时，流量将在 4 台服务器之间进行平均分配，但权重值可以任意进行分配，建议读者通过设置相应权重，尝试为服务器分配不同比例的流量。要使新配置生效，需要执行 shell 命令：/etc/init. d/nginx reload。

⊖ 这一实例来自于在史蒂文生理工学院执教的 CS－524 课程（云计算简介）中的实验任务。作者们将实例的设计归功于一位才华横溢的开发人员——于波（Bo Yu），他是 2014 年春季学期的助教。



5.4.3 使用 DNS 进行负载均衡 ★★★

即使不使用专用的前端服务器，也可以实现负载均衡。在这种情况下，负载均衡是由 DNS 转换功能实体执行的，它将向服务器农场返回特定服务器的 IP 地址。DNS 支持的负载均衡涉及两个问题：①指定服务器的地址；②指定负载均衡算法。

先讨论第一个问题。至少可以采用两种方式来指定服务器的地址，如图 5.36 所示。

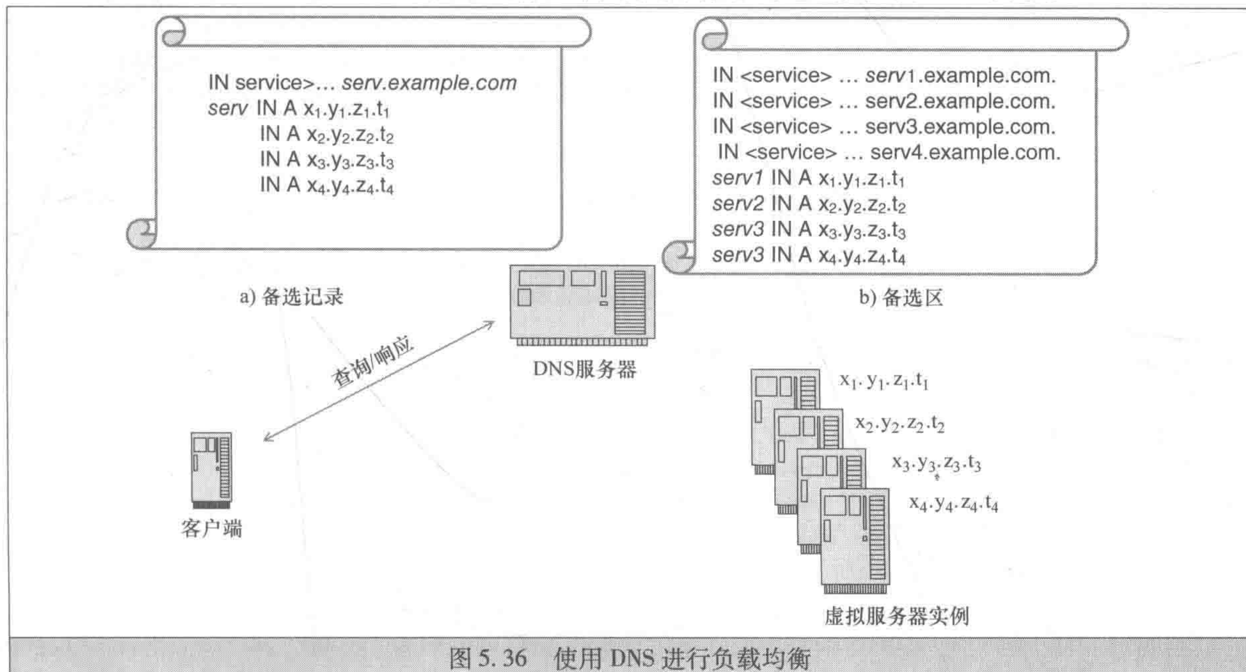


图 5.36 使用 DNS 进行负载均衡

在这两种情形中，处理一种抽象服务 <service>，它可以是客户端和 DNS 服务器支持的任何应用服务（WWW、FTP、Email、SIP 等）。图 5.36a 显示了将 serv. example. com 转换为包含 4 个 IP 地址的列表的响应记录。图 5.36b 所示的备选方案是将 serv. example. com 转换为 4 个区域的授权服务器列表，以便每个区域由合适服务器进行管理以接收负载[㊀]。后一种方法减缓了事务发展速度，但是在处理过载方面具有优势：如果服务器太忙，则它将不会响应 DNS 查询，因而将显示不存在。

在用于选择多个记录返回顺序的算法方面，该算法可以通过使用 rrset - order 规范（在绑定实现方案）来定义。具体来说，排序属性可以取固定值、随机值或循环值（默认设置）。当排序属性取固定值时，记录按照其定义的顺序准确地返回；当排序属性取随机值时，顺序是随机的；当排序属性取循环值时，顺序被循环换位，进而导致轮询调度。

DNS 支持的负载均衡存在的一个问题是缓存——在 TTL 参数指定的持续时间内，缓存返回的响应相同。处理此问题的极端方法是使 TTL 值等于 0，但会增加 DNS 服务器上的负载。缓存问题可能是需要处理的最严重的问题。许多问题也是由服务器的可用性（或者可用性相当差）引起的：如果服务器不可用，则 DNS 服务器可能仍然返回其地址。（如前所述，将服务器与其区域相关联，可以完全解决这一问题。）在不太极端的情况下，虽然 DNS 不知道服务器各自的负载，但它仍只能提供服务分发功能而不是负载分配功能。

为了克服这一困难，需要增强 DNS 服务器的一些功能，使其与每台服务器进行交互，以收集与其各自负载相关的信息，然后基于这些信息做出调度决策[㊁]。此类服务器的早期（大约 20 世纪 90 年代后期）实现方案是由斯坦福大学的 Rob Riepel 开发的 Lbnamed DNS 服务器。

随着时间的推移，DNS 负载均衡功能已经向智能网络处理（见图 5.30）方向演进——由专用服务逻辑引导。当接收到查询消息时，后者在 DNS 服务器上被调用，且能够处理许多要素，包括发出请

㊀ 为了实现这一目标，相关主机也必须运行 DNS 服务器软件（可能是一个非常基本、非常简单的版本）。

㊁ 这种交互没有标准，主要取决于实现方案。

求的客户端地理位置（以便找到最靠近客户端的服务器）⊖。（在和与会 Acamai 代表的密谈中，我们被告知“DNS 服务器拥有智能网涉及的任何东西！”）这再次证明：相对于技术潮流来说，概念是一成不变的——对 PSTN 有益的东西对互联网仍然有益。

市场上存在着相关产品。关于能够证明上述结论的实例⊖，我们推荐读者查阅白皮书[20]。

参考文献

[1] Nabokov, V. V. (1979) The Gift. Collins Collector's Choice: Five Novels. Collins, London, p. 445.

[2] Gabrilovich, E. and Gontmakher, A. (2002) The homograph attack. Communications of the ACM, 45 (2), 128.

[3] Son, S. and Shmatikov, V. (2010) The hitchhiker's guide to DNS cache poisoning. Proceedings of the 6th International ICST Conference on Security and Privacy in Communication Networks (SecureComm), Singapore, September, pp. 466 – 483.

[4] Goldsmith, J. L. and Wu, T. (2006) Who Controls the Internet? Illusions of a Borderless World. Oxford University Press, Oxford.

[5] Scarfone, K. and Hoffman, P. (2009) Guidelines on Firewall and Firewall Policies. Recommendations of the National Institute of Standards and Technology, NIST Special Publication 800 – 41, Revision one. US Department of Commerce, Gaithersburg, MD.

[6] Avolio, F. (1999) Firewalls and Internet security, the second hundred (Internet) years. The Internet Protocol Journal, 2 (2), 24 – 32.

[7] Bellovin, S. M., Cheswick, W. R., and Rubin, A. D. (2003) Firewalls and Internet Security: Repelling the Wily Hacker, 2nd edn. Addison – Wesley Professional, Boston, MA.

[8] Eichin, M. W. and Rochlis, J. A. (1989) With microscope and tweezers: An analysis of the Internet virus of November 1988. www. mit. edu/people/eichin/virus/main. html (presented at the 1989 IEEE Symposium on Research on Security and Privacy).

[9] United States General Accounting Office (1989) Virus highlights need for improved Internet management. Report to Chairman, Subcommittee on Telecommunications and Finance, Committee on Energy and Commerce, House of Representatives. IMTEC – 89 – 57, June 12. www. gao. gov/assets/150/147892. pdf.

[10] P'olya, G. (1945) How to Solve It. Princeton University Press, Princeton, NJ.

[11] Eddie, W. M. (2006) Defenses against TCP SYN flooding attacks. The Internet Protocol Journal, 9 (4), 2 – 16.

[12] Nietzsche, F. W. (1886) Beyond Good and Evil. Cited from the 2006 Filiquarian Publishing edition, New York, p. 82.

[13] Dutcher, B. (2001) The NAT Handbook. John Wiley & Sons, Inc., New York.

[14] Zhang, L. (2007) A retrospective view of NAT. IETF Journal, 3 (2), 14 – 20.

[15] Faynberg, I., Lu, H. – L., and Gabuzda, L. (2000) Converged Networks and Services: Internetworking IP and the PSTN. John Wiley & Sons, Inc., New York.

[16] Vegoda, L. (2007) Used but unallocated: Potentially awkward/8 assignments. The Internet Protocol Journal, 10 (3), 29 – 33.

[17] Faynberg, I., Gabuzda, L. R., Kaplan, M. P., and Shah, N. (1996) The Intelligent Network Standards: Their Applications to Services. McGraw – Hill, New York.

[18] Meyer, T. and Tschudin, C. (2009) A Self – Healing Load Balancing Protocol and Implementation. Technical Report CS – 2009 – 001, University of Basel.

[19] Bourke, T. (2001) Server Load Balancing. O'Reilly & Associates, Sebastopol, CA.

[20] Elfiq Networks (2012) Application and Service Delivery with the Elfiq iDNS Module. Technical White Paper, Elfiq Inc., Montreal. www. elfiq. com/sites/default/files/elfiq _ white _ paper _ idns _ module _ v1. 63. pdf.

⊖ 在讨论内容传送服务时，将重新讨论这一问题。

⊖ 在本书中，并没有试图进行市场或产品调查。

第6章 »
现代数据中心的云存储与结构

数据中心是云计算的主要应用场景。

数据中心是服务器、存储器和通信设备以及必要的实用工具（如电源、冷却和通风设备）驻留的位置。以此种方式来配置设备是顺理成章的，因为环境需求和物理安全需求通常是相同的。这种配置还可以简化操作和维护。举个恰当的例子：在物理上维护1个房间内的10台计算机，比维护分布在5个房间内的10台计算机要容易得多。

大多数数据中心消耗了大量不必要的能源㊀，浪费了从电网输送来的90%或更多的电力[1]。导致效率低下的原因之一是服务器未得到充分利用：典型的利用率变化范围为6%～12%㊁。数据中心的虚拟化可提供一种提高服务器利用率并降低能耗的方法。同时，它还通过硬件、物理外壳与布线、占地面积和其他物理属性来改变传统数据中心的划分方法。

采用此种划分方法得到的虚拟数据中心不再具有明确的物理边界。物理数据中心可以托管多个虚拟数据中心，这一点是不言而喻的。如果虚拟数据中心为多个机构提供服务，则物理数据中心就是多租户的。

但是，也可能存在跨多个物理数据中心的虚拟数据中心。例如，某个机构可能会在维护私有数据中心的同时，将其部分IT（Information Technology，信息技术）基础设施进行外包。在这种情况下，两个数据中心在地理上和管理上是分开的，但可以通过适当的虚拟专用网（或隧道）机制将其拼接在一起，以确保起到隔离作用。

当然，虚拟资源的按需分配和已分配资源的动态重定位是支撑数据中心虚拟化的两大基本特征。在何处分配和重定位资源可能会基于诸如性能需求、负载均衡、改善恢复能力、灾难恢复和法规遵循等标准。从物理资源到虚拟资源的映射是一种与实现相关的问题。总之，除虚拟化技术外，还需要一种用于管理所有跨基础架构资源并为应用提供统一接口的云管理系统[2]。在第7章中，将对云管理系统进行讨论。

本章将对数据中心虚拟化的推动因素进行讨论。首先简要回顾传统数据中心，并介绍其高级功能架构。该架构的核心组件是计算㊂（sic）、存储和网络。传统数据中心的特点之一是使用专业化的专用网络来存储流量以提高性能。这在成本上是不划算的。幸运的是，技术进步可以在不导致性能恶化的情况下，针对所有流量使用单一网络。这样，就出现了将在本章进行描述的下一代数据中心。

然后，重点关注存储相关事宜，因为计算和网络都有自己独立的章节。我们利用全球网络存储工业协会（Storage Networking Industry Association，SNIA）的共享存储模型[3]来介绍存储分类。研究3种类型的存储，它们是根据与主机的连接方式进行区分的：直连式存储（Direct - Attached Storage，DAS）、网络连接存储（Network - Attached Storage，NAS）和存储区域网络（Storage Area Network，SAN）。将处理器和存储设备互连的技术既包括电缆最大长度约为10m的小型计算机系统接口（Small Computer System Interface，SCSI），又包括最大线缆长度约为10km的光纤信道（Fibre Channel，FC）或以太网。长期以来，小型计算机系统接口（SCSI）是用于直连式存储的主流技术。然而，小型计算机系统接口（SCSI）的并行总线设计限制了速度和电缆长度，它已经被串行SCSI（Serial Attached SCSI，SAS）技术所取代。串行SCSI使得互连速度更快，通信距离更远。即使并非不可能，直连式存储要实现共享也是非常困难的。

㊀ 谷歌的数据中心属于例外情况，其耗能比典型数据中心降低了50%以上。这可以通过若干个步骤来实现：将服务器表面温度提高到80℉，使用外部空气进行冷却，并构建效率高达93%的定制服务器。

㊁ 另一个原因是服务器始终处于运行状态的实际做法。

㊂ 计算（名词）是一个业界用于指代计算资源而不是存储资源的新（且相当普通和不精确）的术语。

网络连接存储解除了这一限制条件。特别地，它支持在 IP（Internet Protocol，互联网协议）网络上进行文件共享。但是，对于需要低层次存储访问的数据库应用来说，文件共享并不起作用。此外，存储吞吐量受底层网络介质的限制。

这是存储区域网络（SAN）的切入点。可以对存储区域网络进行调整，以适用于高速互连存储系统，同时它还支持资源池和对资源池的块级访问。它主要基于数据中心的主流技术——光纤信道（FC），它综合了串行 I/O（Input/Output，输入/输出）总线和交换网络的优点。

显而易见，部署和管理一个用于存储的独立定制网络是非常不经济的，因为这需要专门硬件以及负责操作硬件的额外工作人员。因此，人们开始对寻找一种采用单一融合网络来承载所有类型流量（同时能够高效承载与存储有关的流量）的方法产生了浓厚的兴趣。综述该领域的发展现状，并重点关注数据中心的两大主要方法：以太网光纤信道（FC over Ethernet，FCoE）和互联网小型计算机系统接口（Internet Small Computer System Interface，iSCSI）。有趣的是，人们在开发 iSCSI 的同时，用于对象存储的标准接口也开始出现在文献中，它的出现有一个很好的理由。下面将要讨论的对象存储，被视为用于实现 iSCSI 支持的可共享存储承诺中缺失的一环：它能够实现具有精细访问控制功能的主机直接访问。

本章讨论的下一个主题是存储虚拟化，这是一种使应用免受物理存储的底层细节影响的机制。对于云计算来说，网络存储虚拟化非常重要，因为它支持高效的资源池化，并简化诸如快照和迁移等管理任务。

最后，讨论固态存储。关于存储介质，存在着具有不同性能和成本的技术。一种极端情况是随机存取存储器（Random Access Memory，RAM）；另一种极端情况是磁带。中间的情况是相对较新的固态技术——闪存。与 RAM 类似，闪存是基于半导体的，因而不涉及移动部件。闪存比硬盘运行速度快，但比 RAM 成本低。这些优点使闪存技术成为存储层级的可行成员，成为硬盘的有力挑战者。此外，鉴于闪存在随机读取操作中的卓越性能，它已成为云计算的关键。对云计算高性能的追求也推动了诸如 RAMCloud[4] 和 Memcached[5] 等的发展，后面将会对其进行介绍。这两项研发成果都采用了随机存取存储器（RAM），但侧重点各不相同。实际上，RAMCloud 旨在构建一种拥有无限容量的远程缓存，而 Memcached 支持一种简单的键值存储，用于在商用计算机池中的各个存储器单元内缓存任意数据。

限于本书篇幅，我们无法涉及与数据中心相关的一些重要问题，如配置、电源和热管理等。推荐大家阅读本章参考文献［6］来补充了解这些知识。

6.1 数据中心基础

图 6.1 描述了传统数据中心的高级功能架构。硬件模块可以按照标准尺寸组织成一排排支架，以方便部署。本节介绍了关键组件，且一旦虚拟化，它们将成为虚拟数据中心的组成部分。

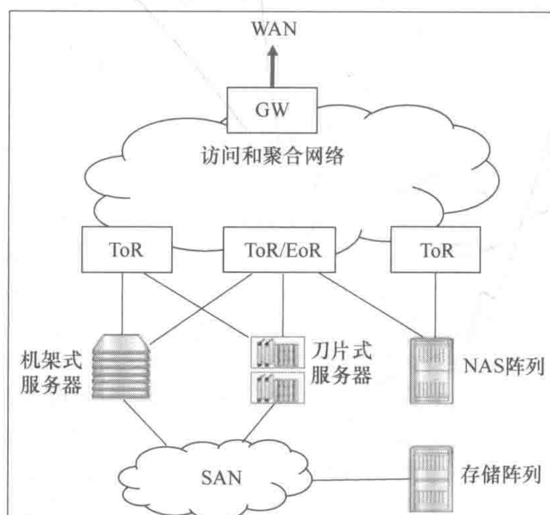


图 6.1 传统数据中心
EoR—行末；GW—网关；SAN—存储区域网络
ToR—柜顶；WAN—广域网

6.1.1 计算 ★★★

计算组件是指可以通过网络进行访问的高性能计算机，称其为服务器。它们是可靠的，且能够处理大量工作负载。服务器在成本和功能方面有很大差别。与台式计算机相比，它们在计算㊀和输入/输出能力方面的可扩展性更强。

服务器可能还具有与桌面计算机不同的外形。它们以机架式或刀片式服务器的形式呈现。可以对这些服务器的外形进行优化，以减小其物理尺寸，降低其互连复杂性（布线难度）。随着服务器数量的不断增加，且这些服务器需要配置在数据中心有限的空间内，因而这种优化是非常必要的。

机架式服务器水平插入机架（通常宽为 19in），它

㊀ 服务器倾向于使用拥有诸如 Intel Xeon 和 AMD Opteron 等多个处理器（或内核）的芯片。

由离散变化的高度来表示，高度值为机架单元（用 RU 或 U 来表示）的倍数。根据本章参考文献［7］的定义，1RU 等于 1.75in（约合 4.445cm）。也就是说，1U 服务器表示高为 1U，2U 服务器表示高为 2U，依此类推。大多数单插槽和双插槽服务器可用作 1U 服务器。

机架式服务器既可以是简单的金属外壳，也可以是包含配电、空气或液体冷却以及键盘、视频和鼠标交换机的复杂设备，该交换机支持单一键盘/视频/鼠标在服务器之间进行共享。

刀片式服务器（或简称为刀片）的结构比机架式服务器更紧凑。较小的外形尺寸是通过去掉与计算不相关的零件（如冷却设备）来实现的。因此，刀片可能只不过是一块拥有处理器、存储器、I/O 和辅助接口的计算机电路板。当然，这样的刀片式服务器肯定不能独立运行。只有将其插入到包含缺失模块的机箱中时，它才能正常运行。机箱内可容纳多个刀片式服务器。它还提供了一台交换机，服务器通过该交换机连接到外部网络。需要注意的是，与机架式服务器类似，机箱也可置于机架中。

当机架空间给定时，它所容纳的机架式服务器要比刀片式服务器多。机架式刀片服务器配置还具有其他优点：功耗降低，布线更加简单，成本降低，等等。这些优点使得刀片式服务器在云计算环境中具有更大的吸引力。

6.1.2 存储 ★★★

根据存储如何连接到服务器，存储可以分为直连式存储（DAS）、网络连接存储（NAS）和存储区域网络（SAN）。为简单起见，图 6.2 只描述了网络连接存储（NAS）和存储区域网络（SAN）。

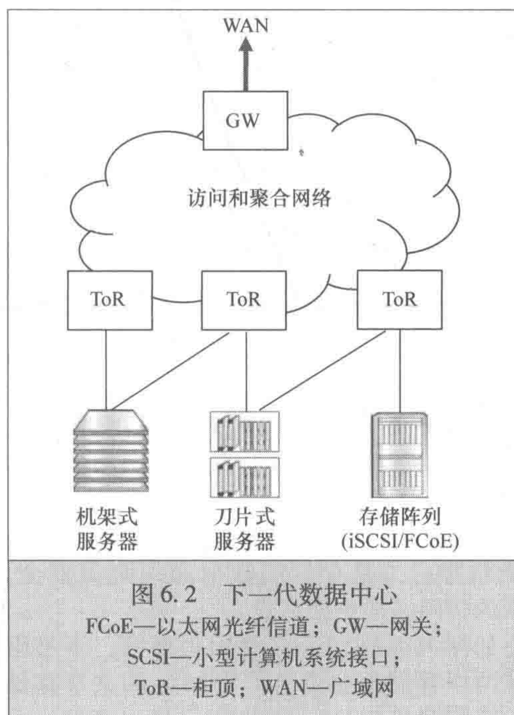


图 6.2　下一代数据中心
FCoE—以太网光纤信道；GW—网关；
SCSI—小型计算机系统接口；
ToR—柜顶；WAN—广域网

顾名思义，直连式存储通过点对点链路直接连接到处理器（这种情形中的主导技术是硬盘驱动器）。相比之下，网络连接存储和存储区域网络存在于网络中。对于存储区域网络来说，这种网络是满足存储流量需求而构建的，且专门用于存储流量。网络连接存储和存储区域网络之间的主要区别在于接口的语义。网络连接存储单元是文件或对象，而存储区域网络单元是磁盘块。另一个主要区别在于底层传输。存储区域网络依靠专门的传输介质——光纤信道进行传输，且为了存储流量，已经对光纤信道进行了优化。除了 IP 网络之外，网络连接存储不需要任何特殊设备。在仔细观察存储形式之后，将讨论两种访问类型的整合问题。

现在，注意到网络连接存储和存储区域网络非常适用于云计算，但直连式存储存在局限性。云计算的重要特征之一是基于资源可用性和地理位置等因素，实现虚拟机的灵活分配。在直连式存储情形中，当虚拟机移动到新的物理主机时，相关存储也需要移动到同一主机，这可能会导致消耗大量带宽和大量时间。

还可以将存储进一步分为在线存储和离线存储。在线存储可以访问服务器，而设计用于归档的离线存储无法访问服务器。磁带库和光盘库是离线存储的常见实现方式。它们通常通过能够定位、取来、安装、卸下和放回磁带或光盘的机器人手臂进行自动控制。例如，谷歌数据中心已经使用机器人磁带库进行备份。

除了磁带和光盘之外，常用的存储介质还包括磁盘和集成电路（即固态电子设备）。其中，磁性硬盘是最流行的[㊀]。考虑到这些硬盘采用的机制，它们更适合顺序访问而不是随机访问。而固态存储克服了这一缺点。固态存储最初用于诸如智能手机、数码相机和 MP3（MPEG Audio Layer 3，音频动态压缩第 3 层）音乐播放器等移动设备，它比硬盘更快（快 100～1000 倍），且比硬盘更坚固，但价格更为昂贵。但是，随着固态存储价格不断下降，它已成为一种可行的选择方案。固态存储特别适用于云计算，

㊀ 自 1965 年以来，硬盘在计算机存储层次结构辅助存储器中也占据着主导地位。磁盘访问速度比 CPU（Central Processing Unit，中央处理器）速度慢得多，相差 7 个数量级，且这一差距还在拉大。

由于它能够实现不相关应用之间的硬件共享，因而使得 I/O 操作比以往任何时候都更随机。本书将在 6.2.7 节中进一步讨论固态存储。

6.1.3 组网 ★★★

数据中心的服务器需要进行互连，且还需要连接到外部世界。随着服务器数量的增加，在给定空间内需要敷设更多的电缆。柜顶（ToR）和行末（EoR）是两种连接方法，采用的连接方法不同，布线方案也会有所不同。在柜顶（ToR）方法中，每个机架顶部都拥有一台交换机，机架中的所有服务器都与该交换机建立连接。因此，将服务器连接到柜顶（ToR）交换机的电缆长度不需要大于机架高度。通常，柜顶（ToR）交换机能够提供外部网络访问功能。一般情况下，柜顶（ToR）交换机拥有能够支持同一机架内服务器的足够端口就万事大吉了。

在行末（EoR）方法中，每排机架的尽头都有一台交换机，该交换机与附近的所有服务器和其他机架中的交换机建立连接。这可能需要在服务器和交换机之间敷设不同长度的线缆。根据机架的实际长度和所需带宽，可能需要敷设光纤（在 ToR 情形中，采用铜线建立连接就足够了）。这里，布线成本可能会超过支持多个链路的服务器成本。

行末（EoR）交换机置于机架中（可能会因其尺寸而单独置于某个机架中）。它可以提供网络访问和聚合。

通常，ToR 和 EoR 交换机采用以太网技术实现。稍后将继续讨论这一话题。现在，注意到，以太网技术对数据中心来说是特别重要的，因为它具备不要求使用独立传输机制（如光纤信道）来进行存储和处理器间通信的潜力。图 6.2 描述了采用普通以太网传输机制的下一代数据中心。

最后，需要注意的是，数据中心聚合网络通过网关连接到广域网（Wide Area Network，WAN）。广域网的另一端可能是单台用户设备或完整的数据中心。

6.2 存储相关事宜

随着连接到互联网的用户和设备数量不断增加，整个世界将被数据淹没。互联网上每天产生的数据流量超过 1 艾字节（即 10^{18} 字节）[⊖]。仅在 2011 年，全球就产生了超过 1.8 亿字节的数据（大约有3/4 的数据由人类用户产生）。数据总量是非常惊人的，而且还在快速增长。对数据的存储和处理需求对云存储产生了巨大的压力。

这一压力主要体现在 3 个方面：一是存储容量需要不断增加；二是存储的数据必须确保安全；三是数据的访问必须更加高效。

为更加深入地研究云存储相关事宜，本书借鉴了全球网络存储工业协会（SNIA）的共享存储模型[3]。该模型最初是于 2001 年开发出来的，反映了存储应作为多个计算系统共享的独立资源进行管理的趋势。十多年后，云计算不仅延续了这一趋势，而且显著放大了这一趋势。

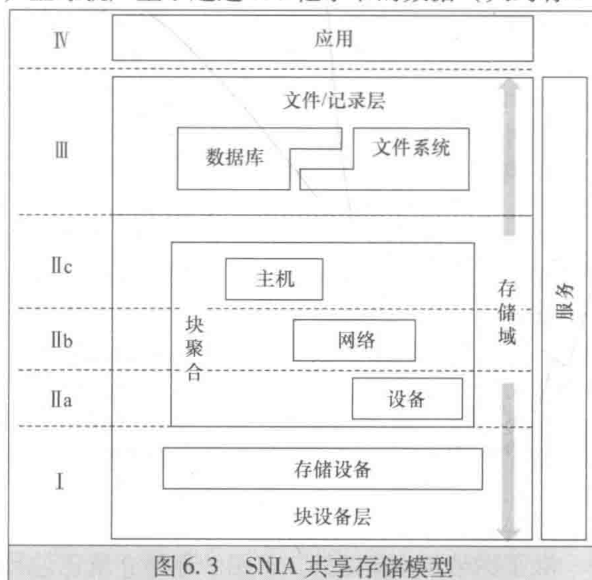


图 6.3 SNIA 共享存储模型

如图 6.3 所示，SNIA 的共享存储模型由多层构成，每层以中立实现方式向高层提供特定服务。这样，可以向高层屏蔽低层的实现细节，且降低了系统的设计复杂性。从这个意义上讲，该模型类似于 OSI（Open System Interconnection，开放式系统互联）模型[8]。

⊖ 标准前缀 exa（艾）是在 1975 年第 15 届国际度量衡大会上被批准的。为了解决“科技进步所需的超大测量尺度”，针对更多字节的前缀［即泽（10^{21} 字节）和尧（10^{24} 字节）］在 1991 年被采纳。按照大数据这一趋势发展，不久将需要超过尧字节的新前缀。



顶部是应用层，它使用的是底层存储域提供的服务。典型的应用实例是 Web 服务器、搜索引擎、分析引擎和在线交易引擎。将应用层包含在模型中只是为了显示存储域与其客户端之间的关系；与存储相关的特定应用有其特殊之处。服务子系统（由图 6.3 中的服务方框来表示）拥有诸如发现、管理、安全和备份等基本功能。

除了服务子系统之外，存储域被划分为文件/记录层、块聚合层和块设备层。文件/记录层为应用层提供服务，一般通过软件来实现。它以文件、文件记录和易于访问应用的类似数据项等形式提供数据。这涉及将可访问应用的数据项映射到底层逻辑构建块上（即逻辑卷㊀）。数据库管理和文件系统在这里得到了广泛使用。文件系统将字节映射到卷中的文件上㊁。类似地，数据库管理系统将记录映射到卷中的表格上。文件/记录层可以在单个主机上实现，也可以作为网络文件系统来实现。后者是网络连接存储的一种特殊情况，稍后将对此进行讨论。

块聚合层为文件/记录层提供服务。它能够提供独立于实际存储设备的块聚合功能，并诠释了设备如何进行互连，以及如何在设备之间分配存储资源。为此，可以通过主机级、网络级或设备级虚拟化来实现聚合，这涉及诸如空间管理、条带化㊂和镜像㊃等任务。特别是将数据输入到存储设备和从存储设备输出数据是由一套外设接口和存储网络标准进行控制的，本书稍后将会对此进行详细介绍。

块设备层为块聚合层提供服务，提供固定大小块的低级存储以及诸如逻辑单元编号、高速缓存和访问控制等功能。

6.2.1 直连式存储 ★★★

直连式存储（DAS）是最常见的存储组织形式，专门适用于单台主机。最初，此类存储所定义的特征包括：①主机和存储设备通过点对点链路进行互连；②主机负责对设备进行控制。图 6.4 给出了一个实例（与 SNIA 模型相关），其中面向块的协议在直接链路上使用，而块聚合过程要么是由主机（通过逻辑卷管理器）实现的，要么是由存储阵列控制器完成的。面向块的协议主要负责处理固定大小块的数据。相比之下，面向文件的协议处理可变大小文件的数据，然后将其分成存储器处理的块。诸如网络文件系统（Network File System，NFS）的网络连接存储采用了面向文件的协议，将在下一节进行详细讨论。

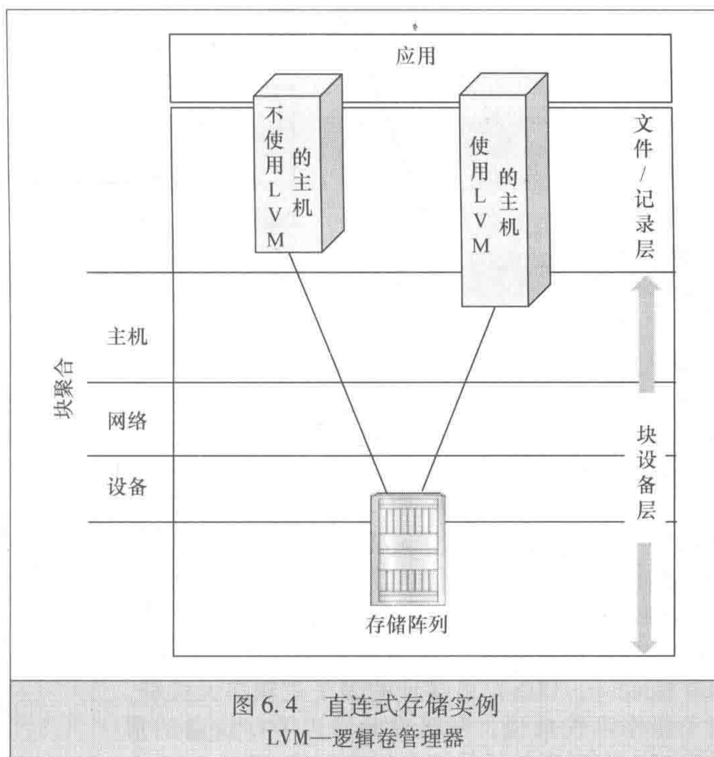


图 6.4 直连式存储实例
LVM—逻辑卷管理器

由于直连式存储（DAS）不易受到网络延迟的影响，因而适用于存储诸如引导映像和交换空间等本地数据。根据存储设备相对于主机的位置，直连式存储可能是内部的，也可能是外部的。主机的内部硬盘驱动器就是内部 DAS 的一个实例。当然，内部 DAS 的总容量在一定程度上受到计算机机箱内物理空间的限制。在这方面，外部 DAS 显得更加灵活。无论 DAS 是内部的还是外部的，主机和存储设备都需要一种用于完成相互通信的接口，以实现 I/O 操作。图 6.5 准确描述了接口所在的位置。主机上的总线适配器和存储设备上的控制器实现了接口功能。主机总线适配器充当系统 I/O 总线和直接连接接口之

㊀ 逻辑卷可以将不连续的物理分区组合起来，并可跨多台物理存储设备。
㊁ 卷是指文件系统的所有块。它可能由一种或多种存储介质的块构成。在前一种情况下，也称其为分区。
㊂ 条带化涉及对序列数据进行逻辑分区和在并行访问的多台设备之间存储分区。这样，会增加总的数据吞吐量。
㊃ 镜像是指在第二组设备中复制数据以实现故障保护。

间的桥梁，并屏蔽掉存储设备的细节。这样，具有标准接口的存储设备可以连接到具有不同处理器和架构的主机。小型计算机系统接口（SCSI）是此类标准接口的一个实例。SCSI 既适用于内部 DAS，又适用于外部 DAS，还经常在数据中心中使用。

1986 年，美国国家标准学会（ANSI）首次对小型计算机系统接口（SCSI）进行了标准化，标准号为 X3. 131 – 1986。该标准基于施加特联合系统接口（Shugart Associates System Interface，SASI），该接口于 1979 年由首家磁盘驱动器制造商[⊖]舒加特联合公司发布。鉴于随后的发展，也将原始标准称为 SCSI – 1。它定义了一种用于连接各类外围设备的并行总线，包括硬盘驱动器、磁带驱动器、CDROM、扫描仪、打印机和主机总线适配器。可以通过菊花链模式将多台设备（或更准确地说是设备控制器）连接到同一总线（见图 6.6），从而形成多点配置。然而，连接的设备数量存在着极限值。极限值取决于数据总线宽度。k 位宽的总线最多支持 k 台设备，包括主机总线适配器。极限值是由于底层设计造成的，该设计将每一个使能位映射到可分配给设备的特定地址。

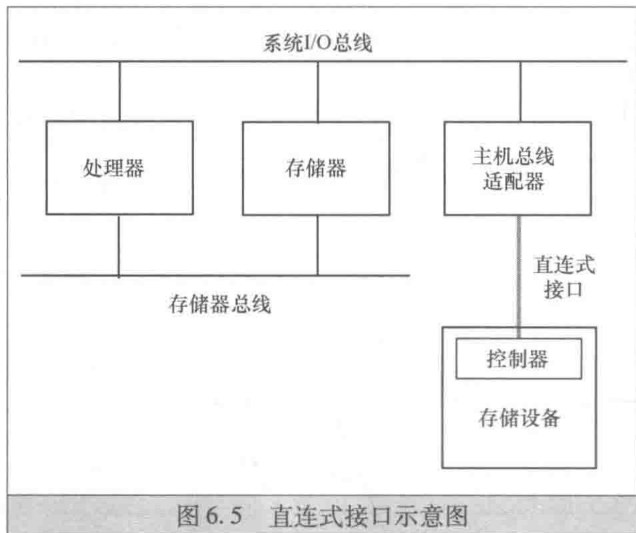


图 6.5　直连式接口示意图

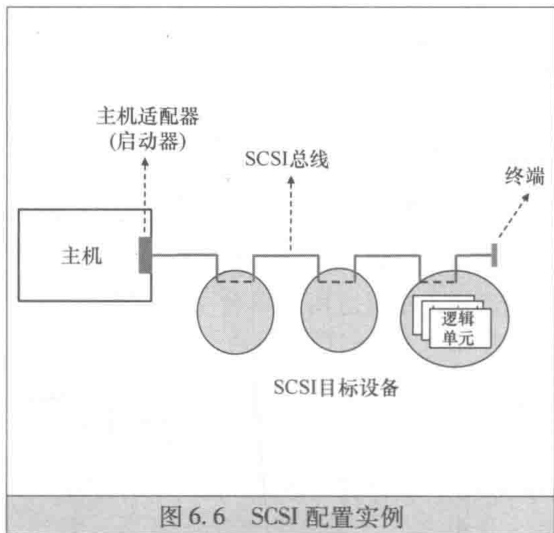


图 6.6　SCSI 配置实例

图 6.7 给出了 8 位数据总线的映射。需要注意的是，SCSI 设备的地址（或标识符）代表着一定的优先级，在竞争过程中，它是总线仲裁的一个重要因素。在总线较宽的情况下，使用相同的优先级方案。也就是说，某个比特越重要，分配给该比特对应的地址优先级越高。事实证明，后向兼容性是非常重要的，因而需要一个稍微复杂的恰当方案来保持总线首个字节的优先级排名。

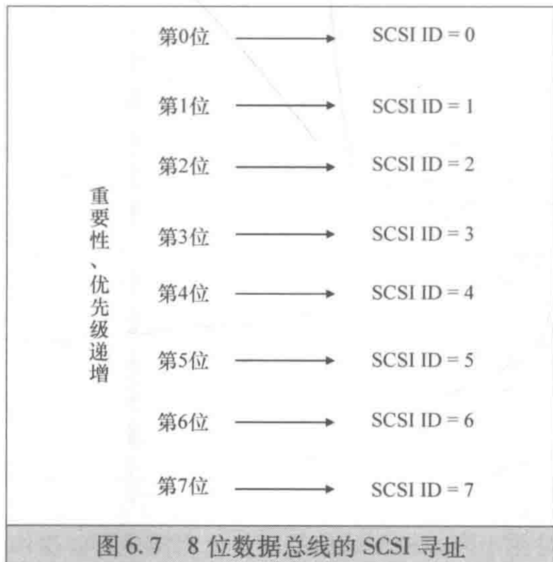


图 6.7　8 位数据总线的 SCSI 寻址

一台设备可以进一步与逻辑单元号（Logical Unit Number，LUN）可寻址的多个逻辑单元关联。对于操作系统来说，逻辑单元号以 I/O 设备的形式呈现。多逻辑单元号设备的实例是 CD（Compact Disc，光盘）自动点唱机，其中每张 CD 都是可独立寻址的逻辑单元。当然，为了给任何 I/O 设备提供服务，同一总线上的设备需要能够相互通信以及与主机进行通信。在高层次上，SCSI 通信基于

⊖ 该接口标准实际上是由国际信息技术标准委员会（International Committee on Information Technology Standards，INCITS）的 X3T9. 2 任务组开发的。X3T9. 2 任务组是 X3T10 技术委员会（Technical Committee，TC）的前身。后来，X3T10 简化为 T10。该技术委员会负责制定 SCSI 规范，包括串联 SCSI 的规范。

如图 6.8 所示的主从式模型。在 SCSI 语法中，我们将发出请求的实体称为启动器，而将响应请求的实体称为目标。请求可以用于诸如读取或写入等 I/O 操作（命令），或者用于诸如中止操作等任务管理功能。在 I/O 操作的情形中，数据传输发生在请求和最终响应之间。在写操作中，数据传输的方向是从启动器到目标；在读操作中，数据传输的方向正好相反。

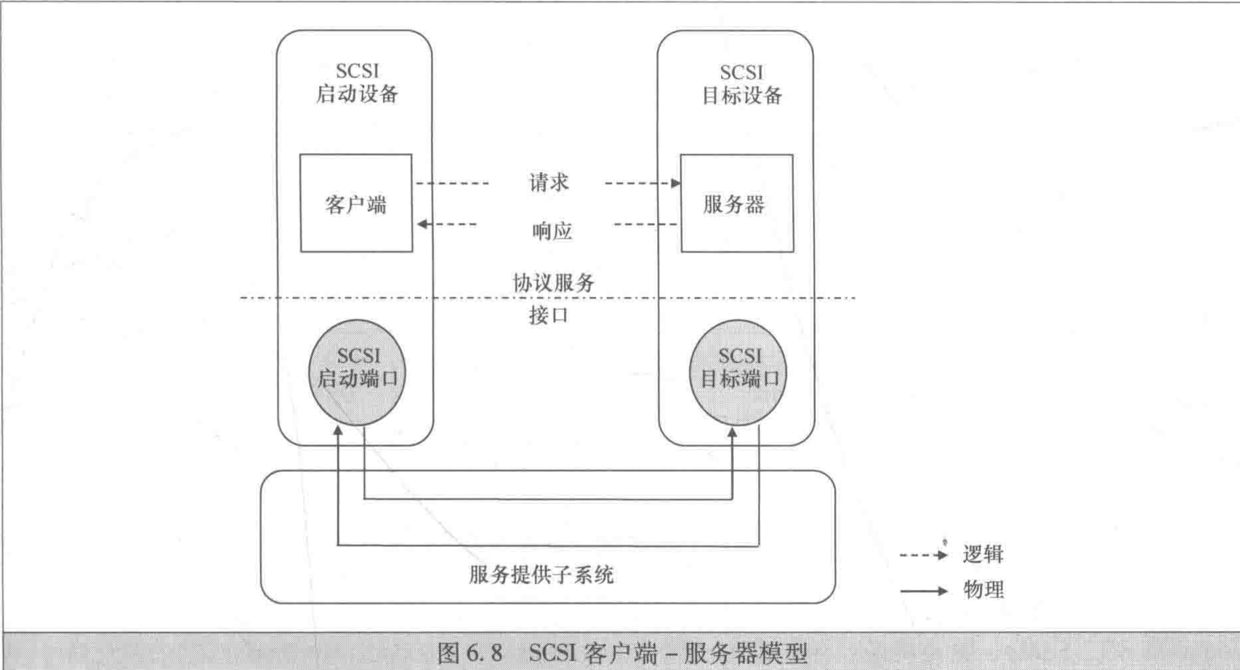


图 6.8 SCSI 客户端 - 服务器模型

SCSI 标准支持一组命令，且为每条命令分配了自己的操作码。命令的细节信息通过总线（它是服务提供子系统的一部分）上的命令描述块（Command Descriptor Block，CDB）传输给目标。由于总线是共享的，因而 SCSI 规定了如何在多个设备之间仲裁总线的排他控制。通常情况下，拥有最高优先级地址的设备将赢得仲裁。赢得仲裁的设备成为启动器，能够选择和指挥目标设备完成所需的 I/O 操作。这样，将可以为主机总线适配器（Host Bus Adaptor，HBA）分配最高优先级标识符，以确保其获得启动器的角色。

多年来，SCSI 已经历了多方面改进（包括可靠性和性能），使得 SCSI 对数据中心越发有用。表 6.1 比较了不同 SCSI 版本在总线宽度、时钟速率、吞吐量和支持的设备数量等方面的性能差别。基于各种改进，SCSI 已从一个涉及诸如协议和布线等各个方面的独立标准，演变成采用分层结构将物理互连与传输协议和 I/O 命令分开的一系列标准。

表 6.1 不同 SCSI 版本的比较

版本	总线宽度/bit	时钟速率/MHz	吞吐量/(Mbit/s)	支持的设备数量
SCSI - 1	8	5	5	8
快速 SCSI（SCSI - 2）	8	10	10	8
快速宽带 SCSI（SCSI - 2）	16	10	20	16
超速 SCSI（SCSI - 3）	8	20	20	8
超速宽带 SCSI（SCSI - 3）	16	20	40	16
超速 2 SCSI	8	40	40	8
超速宽带 2 SCSI	16	40	80	16
超速 3 SCSI	16	40	160	16
超速 - 320 SCSI	16	80	320	16
超速 - 640 SCSI	16	160	640	16

图6.9所示是不断演进的SCSI系列的快照，它涵盖了I/O设备和互连的各种类型。

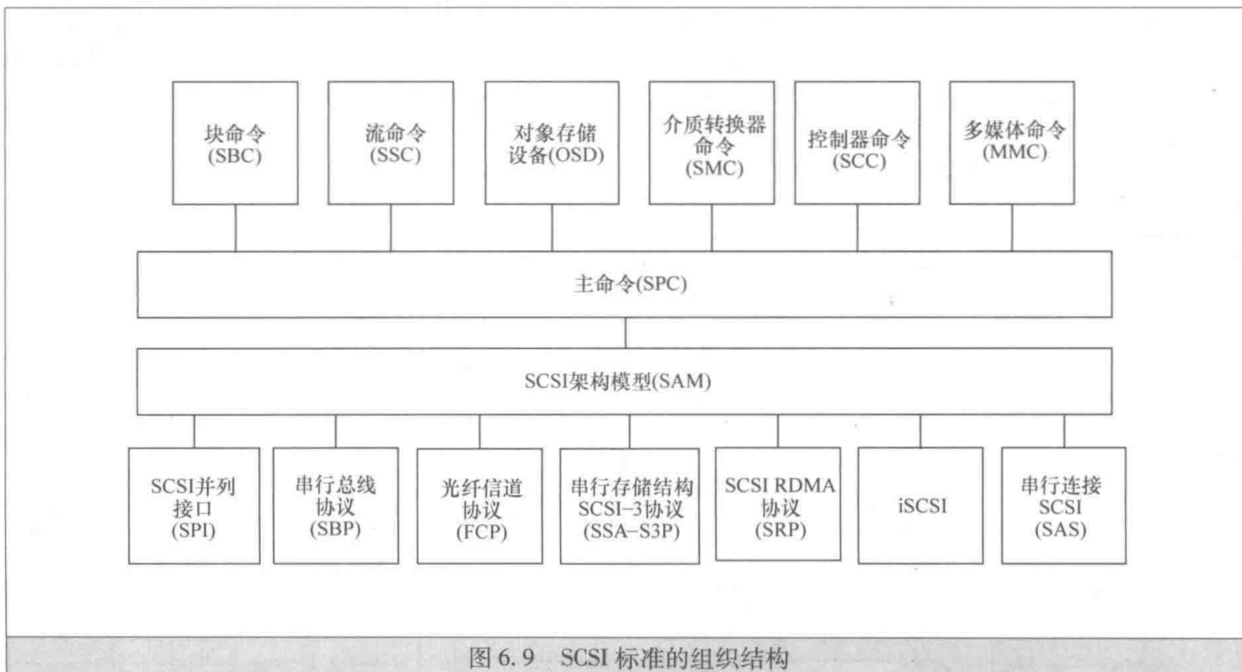


图6.9　SCSI标准的组织结构

架构的中心是SCSI架构模型（SCSI Architecture Model，SAM），这是将系列标准连在一起的纽带。它规定了I/O设备和互连在对象[㊀]、协议层和服务接口方面共同行为的功能抽象。特别地，相邻协议层通过明确的服务请求、指示、响应和确认来进行交互（见图6.10）。分层结构支持接口在实际实现方案中灵活选择硬件、软件和介质，它还支持每一层独立演进。

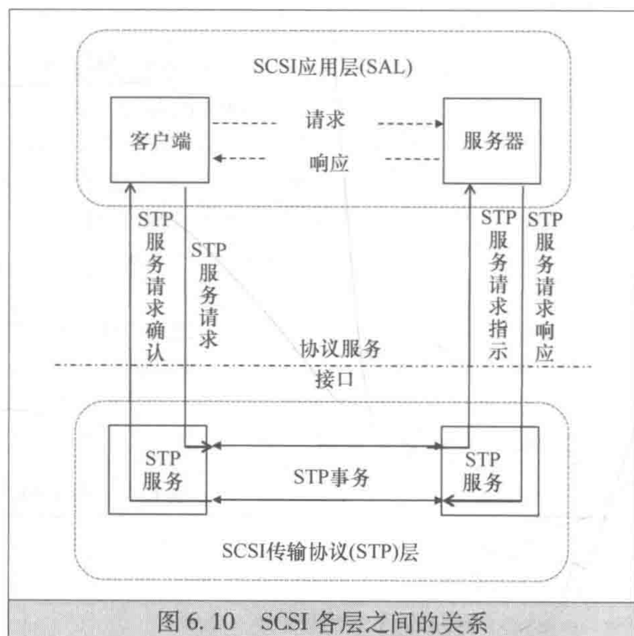


图6.10　SCSI各层之间的关系

作为串行连接SCSI（Serial Attached SCSI，SAS）的一大重要进展，是在2003年首次发布了一项ANSI（美国国家标准学会）标准，标准号为ANSI INCITS 376－2003。当吞吐量增加时，改为采用串行连接能解决并行连接面临的日益棘手的问题。两者的关键区别在于物理层。传统SCSI使用多条线路来并行传输数据，易出现包括串扰和信号到达时间不一致（即定时偏移）等问题。相比之下，串行连接SCSI使用单一线路来依次传输数据。串行连接SCSI解决了并行连接存在的问题，它能够支持更快的时钟和更长的传输距离。在编写本书时，SAS－3版本中的吞吐量已达到1.5Gbit/s，且吞吐量有望继续增长。相比之下，传统SCSI的最快版本（即超速－640 SCSI）支持的吞吐量达到640Mbit/s。同时，串行连接SCSI还具有其他优点，包括更好的可扩展性（即连接成千上万台设备的能力）和简化的布线。由于串行连接SCSI的优势，SAS正在逐步取代其前任——并行连接[㊁]。

㊀　例如，启动器和目标之间的关系由名为nexus的对象来表示。

㊁　“前任”是指SCSI－3。对应的命令集被保存在存储区域网络（SAN）的关键协议栈中，该协议栈是同时开发的。稍后会讨论存储区域网络（SAN）。



实际上，小型计算机系统接口（SCSI）技术的演进遵循了高级技术附件（Advanced Technology Attachment，ATA）的发展路径，这是一种适用于个人计算机（Personal Computer，PC）和电子设备中内部直连式存储的低成本流行接口。最初，ATA 是作为 IBM PC AT 的并行接口设计的，并最终导致串行高级技术附件（Serial Advanced Technology Attachment，SATA）的出现，且串行连接 SCSI 最终采用了 SATA。其中，串行连接 SCSI 使用点对点互连（与 SATA 一样），它支持 SATA 信号的超集（这是通过电力线传输的，用于初始化、复位和速度协商的加电模式），并采用与 SATA 兼容的连接器。因此，实现一组 SAS 和 SATA 设备的混合互连是完全可能的，这反过来又提高了 SATA 技术的相关性以及用于满足数据中心需求的外部存储的后续扩展性（称为 eSATA）。

点对点互连是菊花链模式的一种背离。当要实现两台以上的 SAS 设备互连时，它需要一种称为扩展器的特殊设备。该设备（与充当启动器、目标或兼有两种功能的 SAS 设备不同）本质上是一种支持启动器连接到多个目标的虚拟电路交换机。它通常采用 3 种路由方法来实现这一目标：直接路由、表路由和递减路由。在直接路由中，扩展器识别连接请求的目标是直接进行连接的，并将该请求按某路径发送出去。在表路由中，扩展器将基于路由表（它是由发现过程提供或创建的）将连接请求路由到连接扩展器处。在递减路由中，扩展器使用其他两种方法将尚未解析的连接请求路由到另一个能够解析该请求的扩展器处。

扩展器分为两类：边缘扩展器和扇出扩展器。它们之间的主要区别是可以连接的扩展器数量。边缘扩展器可以连接到多台 SCSI 设备，但只能与另一台扩展器建立连接。相比之下，扇出扩展器可以连接到多台 SCSI 设备和多台扩展器。图 6.11 给出了一种包含 1 台主机总线适配器（HBA）、5 台 SAS 存储设备和 2 台扩展器的示例配置。主机总线适配器（HBA）通过扩展器中提供的专用虚拟电路以点对点的方式连接到每台存储设备。通过增加扩展器数，可以非常简单地实现与更多 SAS 设备的互连。扩展器的使用相对较灵活从而使得串行连接 SCSI 能够支持比传统 SCSI 更多的设备。然而，这些增益是以牺牲简单为代价的。一系列新需求开始出现，包括连接和配置管理的需求、寻址方案健壮性需求（与并行总线的物理布局无关），以及 SAS 设备和扩展器之间的通信机制。

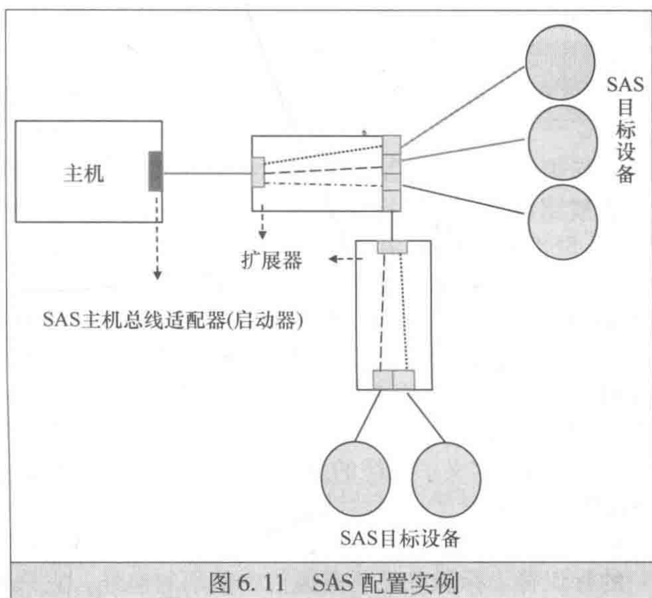


图 6.11 SAS 配置实例

SAS 架构支持 3 种协议：

1）串行 SCSI 协议（Serial SCSI Protocol，SSP），适用于两台 SAS 设备之间以及 SAS 设备和扩展器之间的通信。该协议在增加对多台启动器和目标的支持的同时，保留了 SCSI 命令集。SSP 是串行连接 SCSI 中的主要协议。

2）SATA 隧道协议（SATA Tunneled Protocol，STP），它支持 SAS 启动器设备通过扩展器与 SATA 目标设备进行通信。作为网关的扩展器在发起端采用 STP；在目标端采用 SATA 协议。STP 对 SATA 进行了扩展，以支持多台启动器。

3）串行管理协议（Serial Management Protocol，SMP），用于与扩展器进行通信。它涵盖了发现和配置管理。

图 6.12 给出了 SAS 架构。正如 OSI 模型所描述的，每一层都为上层提供服务，并利用下层提供的服务。

1）物理层负责处理线缆、连接器和收发器的物理和电气特性。

2）Phy 层负责处理线路编码、带外信号和串行传输所需的其他准备工作（如速度协商）。层的名称反映了逻辑构造 Phy，它代表了设备上的收发器（由发送器和接收器组成）。Phy 层拥有设备中唯一



图 6.12　串行连接 SCSI 架构

ATA—高级技术附件；SSP—串行 SCSI 协议；STP—串行 ATA 隧道协议；SMP—串行管理协议

的 8 位标识符。标识符是由管理功能进行分配的。其值是大于或等于 0 且小于设备上物理标识符数的整数。在线路编码中，串行连接 SCSI（SAS）规定了 8b/10b，服从光纤信道（FC）的引导（稍后将进行讨论）。8b/10b 最初是由 IBM 研究人员[9]开发出来的用于高速数据传输的编码技术。名字反映了编码方案的一个关键特征：在传输之前，需要完成数据块从 8 位到 10 位的变换。对该变换进行优化，使得每个编码块中拥有足够的转换（即从 0 到 1 或从 1 到 0），以确保发送方和接收方保持同步，同时尽可能使 0 和 1 的数目相等，以确保直流分量最小。从 8 位到 10 位的扩展为这种优化提供了足够的空间，但同时会产生 25% 的传输开销。相比之下，在 10Mbit/s 以太网中使用的曼彻斯特编码具有 100% 的传输开销。随着速度的增加，大开销成为一个不容忽视的问题。

3）链路层定义了线缆的原型及其编码，以及处理过程（如连接管理和流量控制）。针对 SSP、STP 和 SMP，分别定义了 3 种链路层。

4）端口层主要负责管理端口上的物理特征。端口包含一个或多个物理特征，并由设备制造商分配唯一的标识符。标识符是所有通信中使用的地址。它采用全球通用名称格式，长度为 64 位，光纤信道也支持这一格式（稍后将对其进行讨论）。

5）传输层涉及 SCSI 架构模型（SAM）中定义的传输服务以及串行 SCSI 协议（SSP）、SATA 隧道协议（STP）和串行管理协议（SMP）的成帧问题（包括帧格式）。在串行 SCSI 协议（SSP）情况下，帧格式包含命令描述块（CDB）数据结构和其他结构用来传输与 SCSI 操作相关的信息。

6）应用层支持 SCSI 操作、ATA 操作和 SAS 管理。例如，为了向服务器发送命令，应用客户端能够调用适当的传输服务（通常以过程调用的形式来实现）。

本章参考文献 [10, 11] 提供了有关 ATA、SCSI 及 SATA、SAS 技术的附加信息。

6.2.2　网络连接存储 ★★★

网络连接存储（NAS）能够通过局域网（Local Area Network，LAN）提供文件级或对象级访问。将存储放到网上有利于多台计算机之间的信息共享，并简化相关的存储管理。在这方面，网络连接存储非常适用于云计算，数据中心的高容量、高可用性 NAS 系统的快速发展证实了这一点。

网络文件系统是网络连接存储最早期和最著名的表现形式。通过应用层协议，可以访问任意数量的远程客户端，仿佛这些客户端就是本地的一样。图 6.13 描述了网络文件系统的结构。为了便于解释网络连接存储，我们回顾一下最初由 Sun Microsystems 公司提出的文件系统的概念及广泛应用的 NFS[12]。



操作系统负责维护主机自带的文件系统。在最高级别，文件系统将以文件和目录㊀集（或文件夹）的形式呈现。文件和目录可以进行生成、删除、打开、关闭、读取和写入。它们也可以从一个目录移到另一个目录。大多数文件系统支持分层结构：目录可能有子目录；子目录还可能有子子目录，诸如此类。图6.14给出了这样一种目录。

文件系统以块为单位对空间进行管理，并与直接连到主机的后端存储协同工作。块是一种长度固定的字节序列，可以作为整体对其进行访问。文件能够以块的链接列表形式来实现。对用户来说，操作系统不同，文件的表示方法也不同。早期的操作系统将文件定义为指定格式的记录列表。相比之下，在UNIX（以及Linux）操作系统中，文件只是一个字节序列。

块大小受到底层存储介质的逻辑结构的约束。传统上，可以将其设置为磁盘（这是过去50年的主要存储介质）上可以进行处理的最小单元的整数倍。

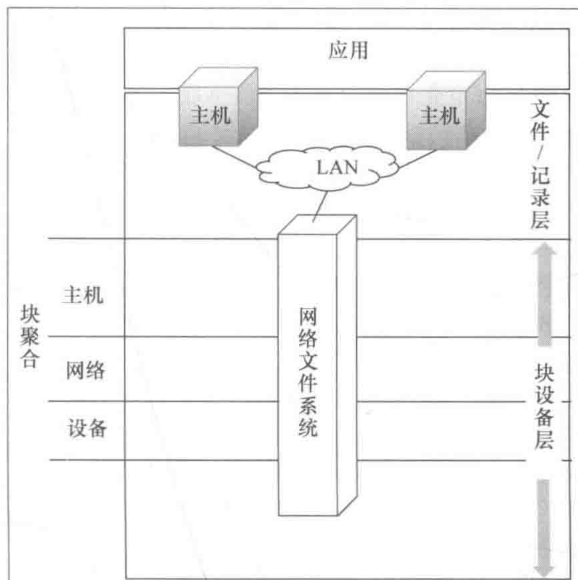


图6.13　网络文件系统

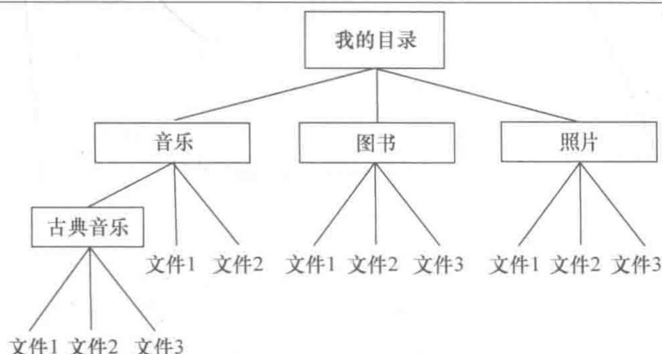


图6.14　分层目录

磁盘是一个盘片栈，每个盘片的两个表面上都涂覆有磁性材料（如氧化铁）。我们也称其为硬盘，因为盘片是由刚性材料制成的。典型硬盘驱动器的结构如图6.15所示。盘片栈在公共轴上以恒定速度旋转，且包含一组读/写头的盘臂（每个表面有一个读/写头）沿着平行的径向线移动。读写操作是通过头部和相关盘片上与其相对的微小表面区域内的涂层材料之间的电磁相互作用来实现的。显而易见，区域越小，磁盘的总容量就越大。物理和工程领域的进步使得面密度稳步提升。特别是，物理学家Albert Fert和Peter Grünberg[13]于1988年独立发现“巨磁电阻”，从而使得面密度超过100Gbit/in^{2}成为可能，并因此获得了2007年诺贝尔物理奖。举例来说，一个内存容量为160GB的iPod经典版只有巴掌大小，而首个千兆字节容量的硬盘驱动器（IBM 3380）却与冰箱一样大。

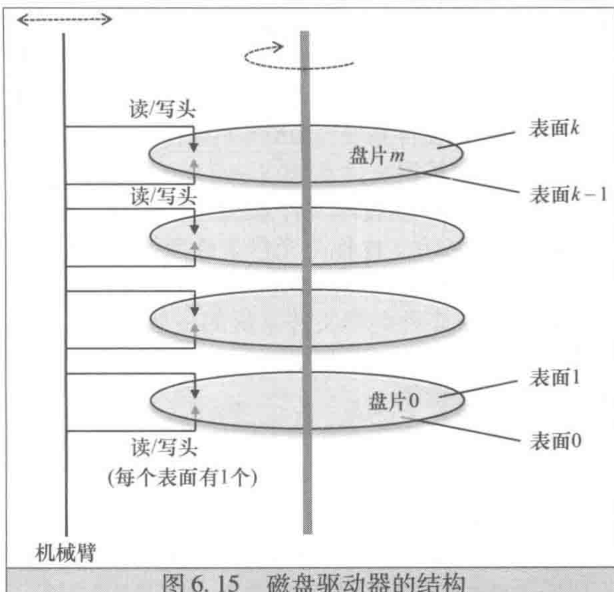


图6.15　磁盘驱动器的结构

㊀　可以将目录视为一种特殊文件，它包含有与文件和子目录相关的信息。

如图6.16所示，可以将每个盘片的表面划分成被称为磁道的同心圆（不同表面上相同半径的磁道集形成一个柱面）。每一磁道依次被划分为数百个扇区。扇区是磁盘上可寻址的最小单位。因此，文件块大小是扇区的倍数。然而，使用的精确块大小存在着折中问题。举例来说，如果块非常大，则大多数文件将绑定多于所需的块，从而浪费存储空间。相反，如果块非常小，则大多数文件将跨越许多块，从而可能会导致不连续问题的发生。所以，对文件的访问受到多次寻道和旋转延迟的影响㊀。此外，非常小的块可能会导致用于跟踪空闲块的数据结构过大。用于跟踪空闲块的通用方案是采用称为位图的特征函数。第 n 位的值表示第 n 个块是处于空闲状态（值为1）还是处于已分配状态（值为0）㊁。大位图、多次寻道和旋转延迟都会降低性能。需要注意的是，谷歌文件系统㊂（Google File System，GFS）[14]使用了64KB的块。该系统选择大块，旨在支持谷歌处理大小超过100MB的数百万个文件的需求。

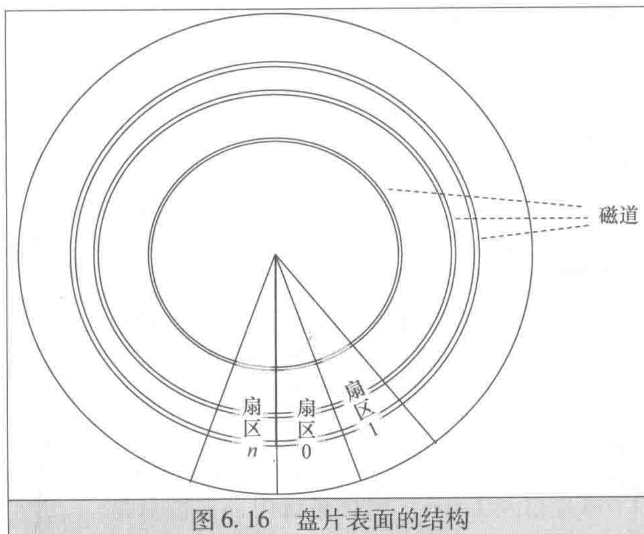


图6.16　盘片表面的结构

为了实现管理的目标，操作系统存储与文件系统相关的信息（如空闲块的类型、布局和位图）以及与每个文件相关的信息（如指向第一个块的指针、文件所有者、最后修改时间和访问权限）。

如果系统在文件更新操作期间崩溃，则文件数据可能受到损坏。现代文件系统管理能够在一定程度上修复损坏的数据。虽然存在着与块和文件一致性相关的算法，但是在大型系统中，运行这些算法需要相当长的时间。幸运的是，还有其他选择（如日志）。这里，系统会在单独的存储区域中保留所有预期的更新日志㊃。

更新要么全部更新，要么全部不更新，必须将其作为原子事务组合在一起。在实际更新进行时，文件系统需要跟踪进度。如果发生故障，则可以在恢复期间使用包含在日志中的信息，通过重复所需的更新或撤销不完整的更新，来修复任何不一致的地方。更恰当地说，称前者为重复日志记录（或新值），后者为撤销日志记录（或旧值）。日志记录是高效的，因为只需要检查最新的日志，而不是检查整个文件系统。如果日志还记录了实际文件的内容，则文件恢复也是可能实现的。毋庸置疑，因日志记录而导致的额外操作可能会对系统性能产生影响。然而，折中方案是大多数现代文件系统支持日志记录功能。德普纳主编的图书[15]对日志记录进行了更为详尽的讨论。

下面列出了常见文件系统的部分实例：

1）UNIX 文件系统（UNIX File System，UFS）；

2）Linux 扩展文件系统（ext2/ext3/ext4）；

3）Windows 新技术文件系统（New Technology File System，NTFS）；

4）ISO 9660（也称为光盘文件系统）。

在这些实例中，ISO 9660尤其引人注目，它不与任何特定的操作系统相关，只是针对操作系统提出了用于支持多种类型文件系统的一般需求。需要注意的是，ISO 9660文件系统的镜像可以通过电子传输作为一个文件（扩展名为.iso）来获取。我们将其称为ISO镜像，这一格式已被用于分发软件模块甚至虚拟机镜像。

操作系统可能直接支持不同的文件系统，而不会试图集成它们。在这种情况下，用户进程可以看到不同文件系统的存在。或者，操作系统可以在文件系统的顶部添加一个抽象层来隐藏其差异，如图

㊀ 将磁头定位在给定柱面上的操作称为寻道，且执行此操作所需的时间称为寻道时间。一旦磁头到达所需的磁道，则需要更多的时间来旋转到磁头下方的准确扇区。我们称这一时间为旋转延迟。

㊁ 位图本身存储在众所周知的位置，并消耗磁盘空间。

㊂ Colossus是谷歌文件系统（GFS）之后的下一代文件系统，充分吸取了GFS的经验教训。

㊃ 存储日志的位置很重要，因为存在着高性能访问的需求。在讨论固态存储时，将再次提到这一主题。

6.17所示。受Sun Microsystems公司首创的虚拟文件系统（Virtual File System，VFS）的启发[16]，许多现代操作系统（特别是那些类似于UNIX的操作系统）都实现了这种层。VFS层为用户进程提供与文件系统无关的接口，支持诸如打开、读取和写入等文件操作过程中的标准系统调用。VFS层还为底层文件系统提供了一种接口。只要底层文件系统支持该接口，则VFS层就不关心这些文件系统的细节，包括文件存储的位置。的确，Sun Microsystems公司原始的VFS包括对诸如网络文件系统（NFS）等远程文件系统的支持，接下来开始对其进行讨论。

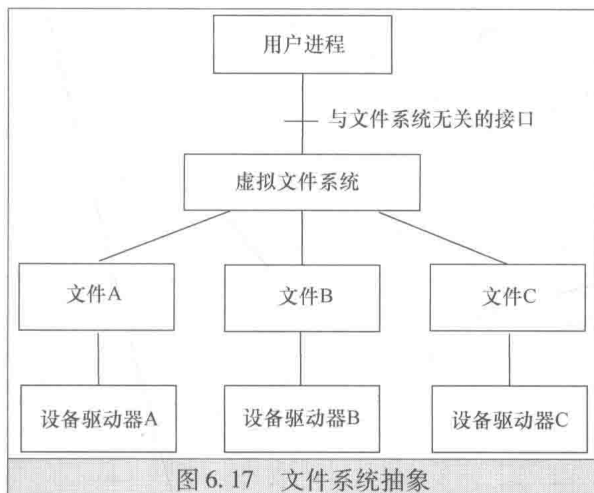


图6.17 文件系统抽象

20世纪80年代，Sun Microsystems公司设计网络文件系统（NFS）的初衷是在安装有不同操作系统的网络计算机之间实现文件共享。图6.18展示了Sun Microsystems公司的实现方案，其中大多数类UNIX操作系统都基本上采用了这一方案。如图6.18所示，NFS通过虚拟文件系统与操作系统进行集成，这是一种天作之合。当用户进程通过系统调用尝试访问文件时，虚拟文件系统（VFS）确定文件是远程的还是本地的。如果文件是远程的，则调用相应的NFS过程。NFS本身是基于客户端/服务器的。NFS客户端通过NFS协议来发起请求，而NFS协议依赖于远程过程调用（Remote Procedure Call，RPC）[17]。NFS服务器只响应请求，自身不采取任何行动。使用RPC可以隐藏与网络相关的详细信息。为了支持不同架构（大端或小端）的机器，RPC又需要一种表示层协议。Sun Microsystems公司的RPC消息协议[18]由IETF（Internet Engineering Task Force，国际互联网工程任务组）进行了标准化，并依赖于外部数据表示（External Data Representation，XDR）标准。与抽象语法标记1（Abstract Syntax Notation 1，ASN.1）类似，XDR[19]是基本数据类型（如字符串、整数、布尔和数组）的一种通用在线表示，它定义了大小、字节顺序和数据对齐。

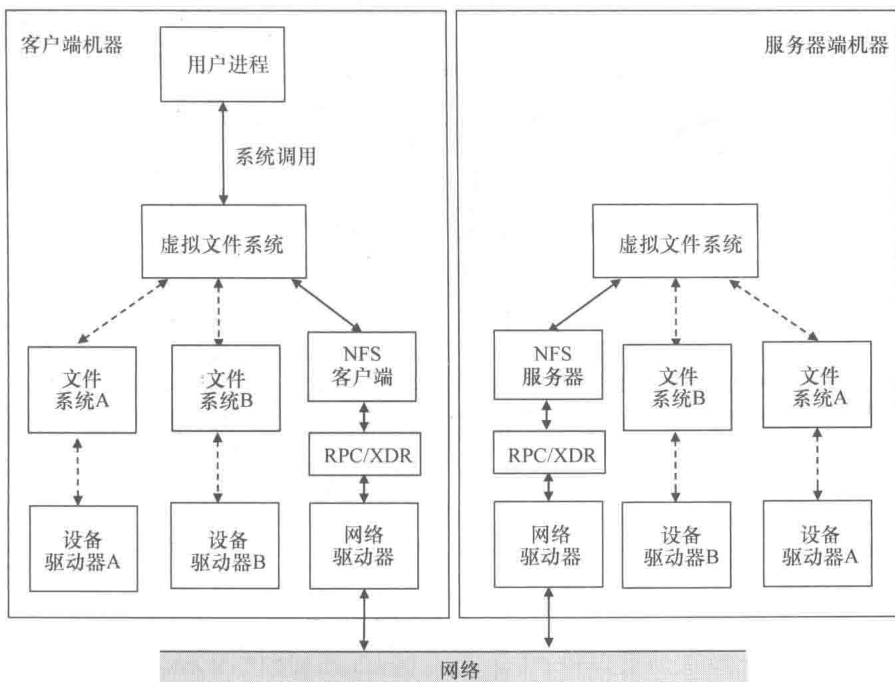


图6.18 NFS的功能视图

在收到 RPC 调用后，服务器会在服务器端机器上调用 VFS 中的相应操作，最终实现本地文件系统的操作。为了返回结果，需要回溯图 6.18 中跨网络的路径。该架构的优点是客户端和服务器端是对称的。因此，在同一台机器上实现客户端和服务器端都是比较简单的。

具体来说，要使远程客户机能够访问文件系统，NFS 服务器应将其导出。要访问远程文件系统的目录，NFS 客户端通过安装协议将其移植（或按 UNIX 语法安装）到本地文件系统。在接收到来自于客户机的安装请求后，服务器端根据预先设置的策略来控制对文件系统的访问，并做出相应响应。一旦客户端收到成功的响应，远程目录即成为本地文件系统的一部分（见图 6.19），且用户进程可以通过常规系统调用来访问远程目录。用于文件访问的客户端和服务器端之间的实际交互是通过 NFS 文件协议实现的。

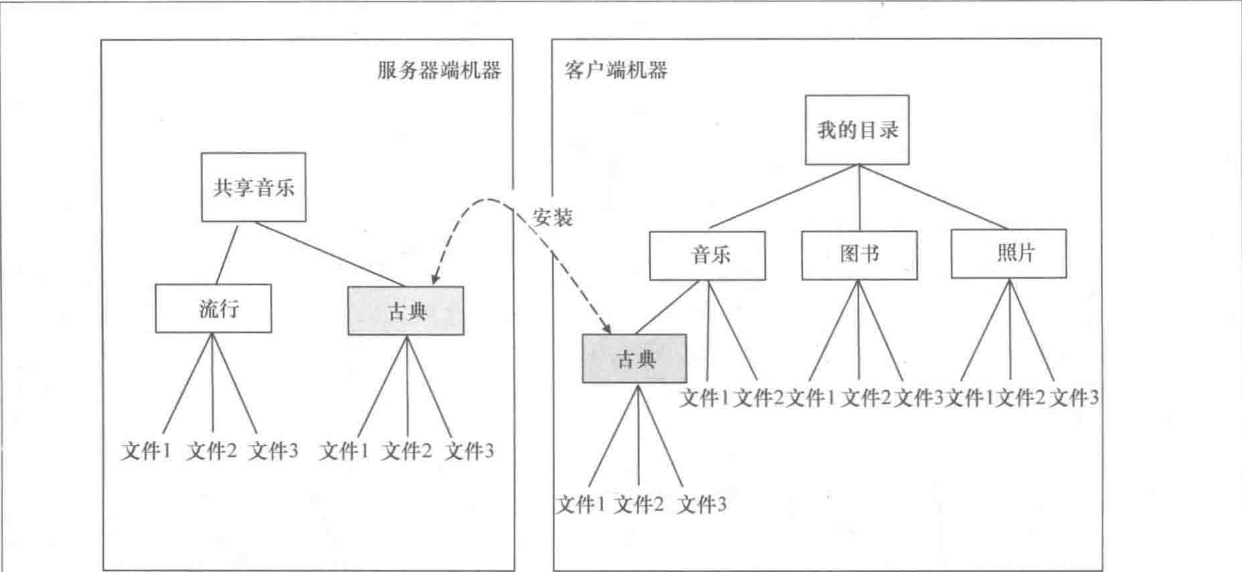


图 6.19 远程文件系统实例

大多数相应的 RPC 例程能够很好地映射到用于文件操作的常规 UNIX 系统调用。例如，图 6.20 显示了远程文件的打开、读取和关闭操作。

一个有趣的细微差别是：关闭系统调用不会导致 RPC 调用。这有两大原因：首先，由于用来为故障发现提供便利的服务器采用的是原始无状态设计（不跟踪过去的请求），因而 NFS 协议没有关闭例程。第二，在这种情况下，不存在文件修改的问题。

即使远程文件操作具有与 RPC 对应的协议，它也不一定会导致 RPC 调用。当信息存储在客户端缓存中时，不需要进行此类调用，这降低了远程过程调用的数量，并提高了系统性能。然而，缓存使得维护文件的一致性变得困难。例如，对某个站点处的文件写操作，可能在其他站点打开并读取此文件时变得无法正常显示。

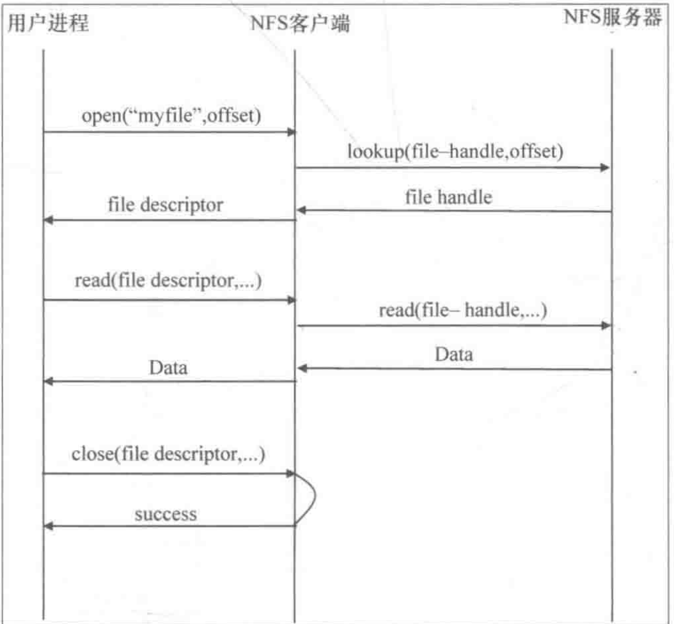


图 6.20 通过 NFS 实现远程文件操作的实例

NFS 自出现以来，经历了数次更新。在这一过程中，无状态设计的限制条件已经放宽，文件一致性得到改善，安全性得到加强。本章参考文献［15］对 NFS 及其演变进行了深入的阐述。

在结束NAS主题讨论之前，观察到它的实现形式通常为独立磁盘冗余阵列（Redundant Arrays of Independent Disks，RAID）。采用RAID技术支持从多个单独磁盘故障中进行恢复，并从总体上提高NAS的性能和可用性。

6.2.3 存储区域网络 ★★★

从历史的角度来看，直连式存储（DAS）是一种烟道式技术，这使其难以共享存储资源和存储信息。网络连接存储（NAS）缓解了这一问题，但仍有改进的空间。特别是存储吞吐量受到所采用的特定网络技术的限制以及块级I/O访问不可用的问题。这就是存储区域网络（SAN）产生的背景。实质上，SAN是一种专门用于实现存储系统互连的高速网络，它支持资源池和对资源池的块级访问。SAN主要基于光纤信道（FC），这是一种综合了串行I/O总线和交换网络优点的标准技术。总线的优点是（反映在“信道”一词的选择中）允许主机查看通过光纤信道以本地连接形式相连的存储设备，并拥有小型计算机系统接口（SCSI）可靠传输的特性。网络的优点是支持多个协议的灵活性和存储设备远距离的动态连接。

美国国家标准学会（ANSI）对光纤信道标准的制定始于1988年，最终形成了ANSI X 3.230－1994规范。国际信息技术标准委员会（INCITS）的T11技术委员会与T10技术委员会（主要负责与FC密切相关的SCSI项目标准化）一起，继续推进光纤信道的标准化工作。

作为一种开放标准，光纤信道（FC）的分层结构如图6.21所示。FC－0定义了采用不同速率进行串行无损耗传输（误码率低）所使用的收发器、连接器、电缆的物理和电气特性。迄今为止，最快的光纤信道（称为32G FC）在每个方向上支持3.2 Gbit/s的传输速率。它支持光纤和铜缆两种类型，甚至可以在端对端路径中采用铜线和光纤混合线缆[㊀]。



图6.21 光纤信道结构

FC－1涉及线路编码和相关的收发器操作。特别是它定义了适用于高速数据传输的一系列编码方案，包括256b/257b、64b/66b和8b/10b。通常情况下，这些编码方案支持接收端的时钟恢复，使其能够在发送和接收期间检测到误码，并有助于实现传输块对齐。您可能会记得采用8b/10b编码方案的串行连接SCSI（SAS）。该编码方案具有25%的传输开销，因而对于更快版本的光纤信道（FC）而言效率不够高。表6.2给出了迄今为止每种光纤信道版本所采用的编码方案。简而言之，64b/66b编码方案（也适用于10G和100G以太网）在传输之前，将每个64位块转换为66位块。256b/257b方案建立在64b/66b方案的基础之上，只是在传输前将每4个64b/66b位块进一步转换为257位块。FC－1还定义

㊀ 科技术语中的拼写选择（fibre而不是fiber）旨在传达这种广泛布线支持的含义。

了多个有序集（即某些编码比特模式）。其中包括用于标记帧边界的帧分隔符，以及用于表示端口是否准备好发送和接收的原始信号。

表 6.2　光纤信道和线路编码

光纤信道版本	编码方案	线路速率/GBaud	吞吐量/(Mbit/s)
1GFC	8b/10b	1.0625	100
2GFC	8b/10b	2.125	200
4GFC	8b/10b	4.25	400
8GFC	8b/10b	8.5	800
10GFC	64b/66b	10.53	1200
16GFC	64b/66b	14.025	1600
32GFC	256b/257b	28.050	3200

FC－2 层由 3 个子层构成，分别是物理子层（FC－2P）、复用子层（FC－2M）和虚拟子层（FC－2V）。FC－2P 涉及帧格式（物理链路上传输信息的基本单位），以及与发送和接收帧有关的事项（如每条链路的流量控制）。帧格式包括用于检测和纠正传输差错的循环冗余校验（Cyclic Redundancy Check，CRC）字段[⊖]。

流量控制机制能够防止发送机向接收机发送超过接收机处理能力的帧。它需要一种反馈机制来支持发送机来调节其传输。如果发送的帧过多，则接收机将丢帧。丢弃的帧需要进行重传，这会加剧拥塞。考虑到无损帧传输的要求，流量控制在光纤信道中尤其重要。为此，光纤信道采用一种基于信用概念的流量控制机制。信用是接收机端用于接收帧的最大缓冲区数[⊜]。在登录过程（稍后进行讨论）中，相关端口（即发送机和接收机）之间需要协商缓冲器到缓冲器（即每条链路）或端到端的信用。当且仅当接收机拥有可用缓冲区时，发送机才能确保其发送帧。在每条链路流量控制（适用于交换网端口）的情况下，发送机通过来自接收机的原始信号（即 R_RDY）的帮助来实现这一目标。信号表示接收机已准备好使用可用缓冲区来接收帧。发送机跟踪可用缓冲区的数量，在发送帧时将其递减 1，并在接收到一个 R_RDY 时将其递增 1。

FC－2M 涉及端到端连接、寻址和路径选择。它支持 3 种类型的连接：点对点、交换网和仲裁环。点到点拓扑是最简单的，两个端口之间存在一条直接链路（类似于前面讨论过的 SAS 端口）。它与直连式存储（DAS）具有相同的效果，同时支持更长的距离和更高的速率。

交换网拓扑最灵活。它包含了一组与网络相连的端口，该网络通过单独的物理链路实现了光纤信道的互连，如图 6.22 所示。根据域和区域，交换网络（或交换网）采用了分层结构和 24 位地址空间。在交换网登录过程（稍后将进行讨论）中，为每个连接端口分配唯一的地址。确切的地址通常取决于交换网（更准确地说，是交换机）上用于连接的物理端口。交换网基于每个帧头中的目标端口地址单独路由帧。

最后，仲裁环拓扑在无须交换网的情况下，支持 3 个或更多个端口进行互连。图 6.23 给出了一个实例，以及采用能够简化布线的集线器（一种不具备任何环路控制功能的简单设备）的另一种方案。在仲裁环上，只有两个端口可以通过仲裁在任何给定时间内进行相互通信。

在所有 3 类拓扑中，通信既可能是单工的，也可能是全双工的，还可能是半双工的；端口可能位于主机总线适配器（HBA）、存储设备控制器、集线器或交换机上。

在交换网拓扑的情形中，可以采用 ANSI INCITS 461－2010 中定义的交换网最短路径优先（Fabric Shortest Path First，FSPF）协议来选择光纤信道上的路径。FSPF 类似于 IP 网络中常用的标准开放最短路径优先（Open Shortest Path First，OSPF）路由协议，是一种链路状态路由协议。

⊖ 规定的循环冗余校验（CRC）算法是光纤分布式数据接口－介质访问控制中指定的帧校验序列（Frame Check Sequence，FCS）。

⊜ 缓冲区是一种能够保存单个帧的逻辑结构。

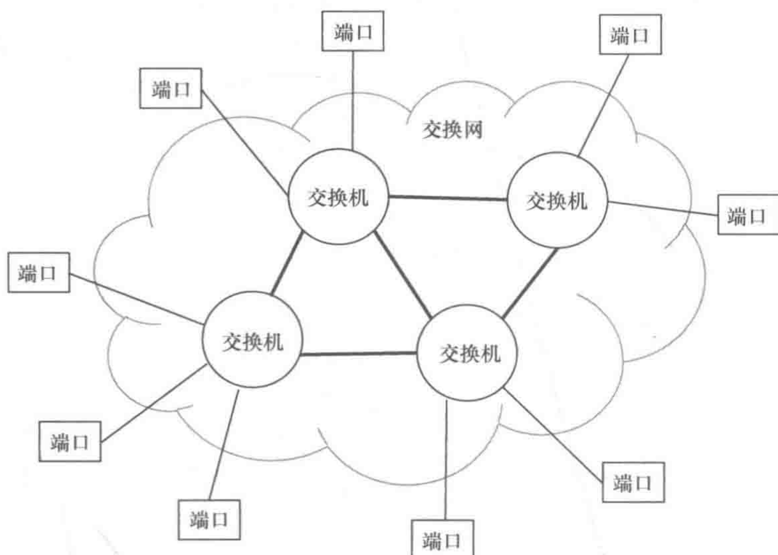


图 6.22　交换网拓扑实例

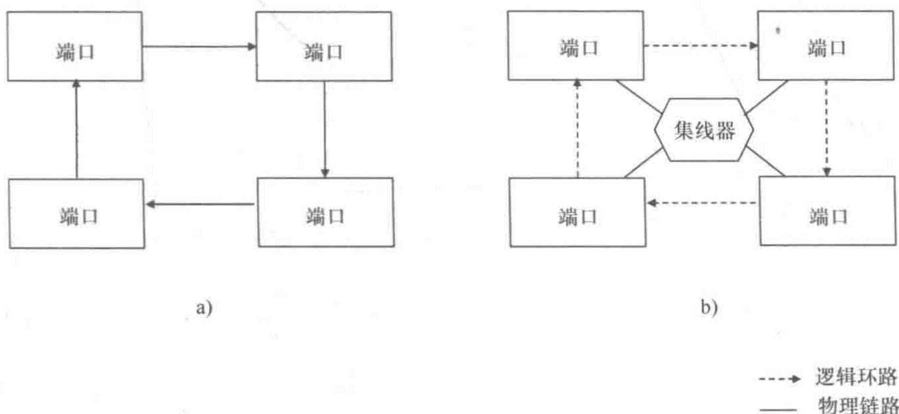


图 6.23　仲裁环实例

通过采用 FSPF 协议，交换网中的交换机能够跟踪所有交换机之间连接的状态，并始终维护交换网的最新拓扑。链路状态信息包括与每条链路相关的成本，它与链路速率成反比[⊖]。基于链路状态和拓扑信息，每台交换机计算到其他交换机的所有可能路径的相应总成本，并选择成本最低的交换机。某条路径的总成本只是其中所有链路的成本之和。图 6.24 给出了一个实例，其中交换机 A 和交换机 C 之间的最低成本路径是通过交换机 B 的。

显而易见，选定的路径只对给定拓扑有效。只要拓扑发生变化，交换机就必须重新进行路径计算。也就是说，如果取消从交换机 A 到交换机 B 的链路，那么路径重新计算将产生两条成本相同的路径。在这种情况下，需要一台断路器，主要是出于负载均衡方面的考虑。

FC－2V 涉及服务等级、端到端流量控制、命名方案以及支持上层协议的分段和重组等。在编写本书时，它支持 3 个服务等级，分别是确认帧传输（第 2 级）、非确认帧传输（第 3 级）和交换机间帧传输（第 F 级）。第 2 级和第 3 级都是数据报服务。帧在无须任何传输次序保证的情况下，通过交换网分别进行路由。第 2 级支持发送帧传输状态的通知消息，而第 3 级则不支持这一功能。

传输确认支持端到端的流量控制，并能改进差错处理。端到端流量控制也是基于信用的，由通信

⊖ 根据 ANSI INCITS 461－2010，链路成本 $= S \times \left(\frac{1.0625 \times e^{12}}{\text{链路波特率}}\right)$，其中 S 是强制定义的因子。默认情况下，将 S 设置为 1.0。

端口在登录时进行协商。当传输确认数显著超过协商的信用值时，发送端可能不会再发送帧。需要注意的是，传输确认在帧级（通过 ACK_1 帧）完成，且开销比在原始信号级完成的确认更大。

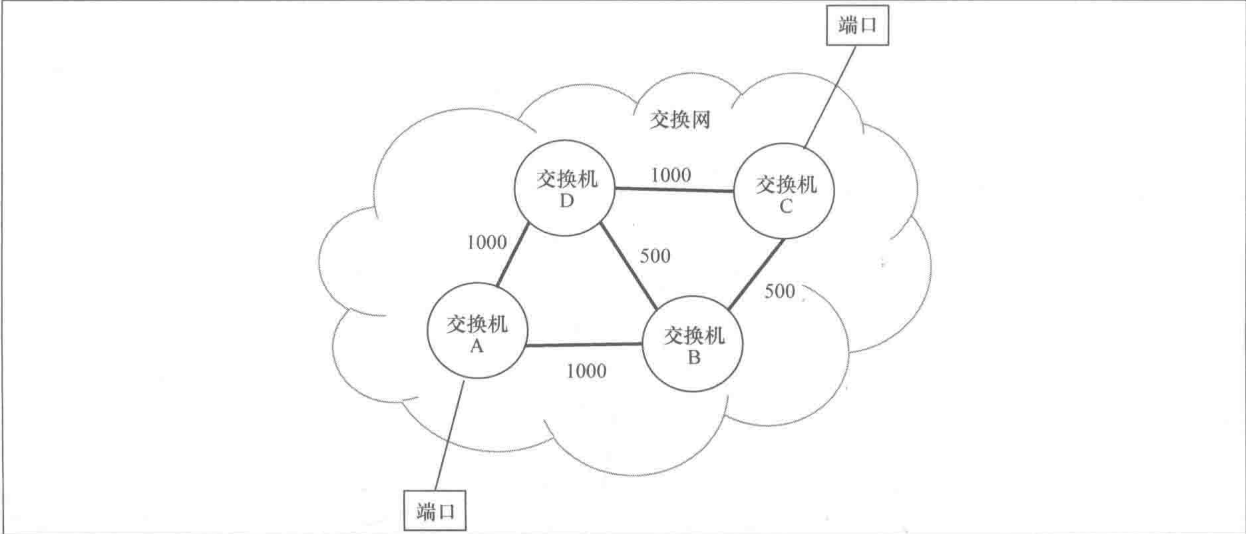


图 6.24　加权路径网络

命名机制类似于光纤信道（FC）中的寻址机制。目前，人们已经提出了若干种用于识别端口、节点（如存储设备和主机总线适配器）、交换网和其他光纤信道实体的方案。在实践中，64 位全球唯一名字（World Wide Name，WWN）就是一种实现方案。WWN 类似于 MAC（Medium Access Control，介质访问控制）地址，它们是由 IEEE（Institute of Electrical and Electronics Engineers，电气和电子工程师协会）以相同的方式进行管理的。全球唯一名字由制造商分配给光纤信道实体。

从各方面考虑，全球唯一名字可以为光纤信道路由提供基本依据。但是，如前所述，光纤信道路由基于一种特殊的 24 位端口地址。出现这种情况的原因是担心 WWN 地址越长，会导致光纤信道路由太慢而无法实现整体性能目标。

因此，端口以两个标识符作为结束。在 FC 语法中，地址标识符的长度是 24 位；名称标识符的长度是 64 位。它们的用途不尽相同。名称标识符在诸如分区等服务中非常有用。基于其名称标识符，可以将 FC 设备分组为多个隔离区，这样跨隔离区的设备不能相互查看和通信。分区可能基于地址标识符，但要对其进行管理比较困难。与名称标识符不同，当设备对交换网上的连接端口进行更改时，地址标识符容易随之发生更改。从安全角度来看，根据地址标识符进行分区所产生的约束条件反而是一件好事。

FC－3 涉及为 FC－4 提供的链路服务，用于管理 FC 设备之间的通信以及交换网与连接设备之间的交互。FC－3 提供了一种链路服务，可以将其进一步细分为基本服务和扩展服务。

基本链路服务集非常小，它旨在支持中止某个序列（该序列是由单个上层协议数据单元分片引起的帧流量），并将结果通知给服务请求方（即请求完成或拒绝）。

相比之下，扩展链路服务集要大得多。特别是它支持登录和注销过程。

登录过程是强制性的。它由两个步骤构成：交换网登录和端口登录。FC 设备上的端口必须首先完成交换网登录步骤，然后再尝试执行其他操作。该步骤涉及将包含交换网登录命令和其他信息的帧发送给已知的端口地址。如果该步骤成功完成，则设备端口会发现拓扑类型（如交换网或点对点），此时将为设备端口分配一个地址（如果交换网存在）。该过程的其余部分需要处理一些服务参数（如支持的服务等级和每条链路流量控制的信用值）。然后，完成端口登录步骤，主要是处理其他服务参数值（如端口名称和端到端流控制的信用值）。

登录过程会导致会话时间过长。只要会话处于激活状态，则需要在两个端口之间执行 I/O 操作。否则，必须建立一次新的登录会话。

注销过程只是终止登录会话，并释放其占用的资源。尽管“登录”一词给人以需要进行身份验证的错觉，可是登录过程却不拥有内置身份验证功能。传统上，FC 设备是可信任的。

最高层 FC-4 涉及桥接下面的传输层以及上面的应用层，使得 FC 设备能够以透明的方式来访问应用。例如，SCSI 感知应用可以访问 FC 设备而不需要进行任何修改。为此，FC-4 定义了应用协议和底层 FC 结构之间的映射。该映射与特定应用协议有关。目前，人们已经定义了一组与特定应用有关的映射，特别是 SCSI 的光纤通道协议（FCP）。正如 SCSI 架构模型中所定义的（前面已经进行了讨论），FCP 能够提供传输协议服务。举几个例子，它涉及与 SCSI 操作（如 CDB）、地址映射和能力发现有关的信息封装。通过 FCP，FC 存储设备可以作为应用的 SCSI 设备出现。

已经定义的其他映射还包括虚拟接口（Virtual Interface，VI）架构的 FC 虚拟接口（FCVI）[20] 和 IPv4/IPv6 的 RFC 4338。虚拟接口架构旨在提供拥有高性能需求的进程。在不涉及特定操作系统服务的情况下，它需要一种受保护的、可直接访问的网络硬件接口。除了传统的发送和接收消息传递结构，它还支持远程直接内存访问（Remote Direct Memory Access，RDMA）。人们已经将 RDMA 应用于分布式科学领域，并证明它是有效的。然而，实践中使用的版本与 VI 架构中的 RDMA 不同。InfiniBand™是一个显著的例子。根据云计算，在虚拟机中设计支持 RDMA 的有效机制是当前的研究热点。

6.2.4 SAN 和以太网的融合 ★★★

光纤信道（FC）存储区域网络（SAN）因具有卓越性能而在数据中心得到广泛应用。但是，需要为存储部署和管理单独的定制网络是一大缺点。这样做需要采购专门的硬件（即定制的 HBA、连接器、电缆和开关）以及专门操作人员的参与。存储区域网络和以太网的融合主要研究如何将以太网应用于所有类型的流量（包括存储）。关键的动机是它拥有降低资本和运营支出的前景。

读者可能记得在第 2 章中，IT 转型和网络功能虚拟化也是同样的动机。

SAN 的分层结构有助于实现如图 6.25 所示的融合方法。这些方法对应于 FC 协议底层模块的交换。在极端方法（即方法 3）中，已经没有任何 FC 的踪影[⊖]。

SCSI命令			
iSCSI	FCP	FCP	FCP
	FCIP	FC-3	FC-3
TCP	TCP	FC-2V	FC-2V
IP	IP	FCoE	FC-2M
			FC-2P
IEEE 802.3 MAC	IEEE 802.3 MAC	IEEE 802.3 MAC	FC-1
IEEE 802.3 PHY	IEEE 802.3 PHY	IEEE 802.3 PHY	FC-0
方法3	方法2	方法1	

图 6.25 融合存储协议选项实例

对光纤信道（FC）来说，由 INCITS T11 技术委员会开发的另两种方法更为实用一些。这个想法是保持各种模块来帮助过渡到新的部署。3 种方法的共同之处在于使用 IEEE 802.3 MAC 层和物理层来代替 FC-1 和 FC-0。方法 1 进一步要求使用 FCoE 层来替换 FC-2M 和 FC-2P，方法 2 使用 TCP/IP 及其上面的 FCIP 来替代 FC-2P、FC-2M、FC-2V 和 FC-3，方法 3 使用 TCP/IP 及其上面的 iSCSI 替换 FC-2P、FC-2M、FC-2V、FC-3 和 FCP。本节的其余部分重点关注两种主要的融合方法——方法 1（FCoE）和方法 3（iSCSI）。

光纤信道（FC）的主要特点是它不会因缓冲区拥塞而导致帧丢失。方法 1 依赖于无损的以太网链

⊖ 这就解释了为什么是由 IETF（而不是由 INCITS T11 技术委员会）来负责相关标准。

路来保持此特性。为此，以太网的暂停机制是至关重要的[1]。该机制支持处于拥塞状态的以太网交换机请求相邻交换机（通过暂停帧）在特定持续时间内不发送帧。如果拥塞持续时间超出请求的有效期限，则交换机可以发送一条新的暂停请求。相反，如果拥塞早于预期得到缓解，则交换机可以通过发送一条持续时间设置为 0 的新暂停请求来取消未完成的请求。暂停机制是基于每条链路开展工作的。其调用是由每台交换机基于本地负载条件独立进行处理的。此外，一旦调用，暂停机制适用于同一链路上的所有流量。

当不同类型的流量共享融合网络时，暂停机制的工作效果不够理想。即使暂停是由与存储无关的流量引起的，存储性能也将受到影响。

幸运的是，存在着一种补救措施，这就是 IEEE 802.1Qbb 中规定的基于优先级的流量控制（Priority - based Flow Control，PFC）机制。PFC 机制支持将暂停机制单独应用于不同优先级的流量。流量优先级划分是通过 IEEE 802.1Q 中定义的 3 位标签来实现的。通过更为精细的控制粒度，不同等级的流量（由于受到标签长度限制，因而最多 8 个等级）不会产生相互干扰。同时，暂停非存储流量以支持存储流量通过并协助满足存储性能要求也是可以实现的。

除了无损以太网之外，方法 1 需要一个新层，即光纤信道 - 主干 5（FC - BB - 5）[2]规范中定义的以太网光纤信道（FCoE）。FCoE 的任务是弥补采用以太网来传输而导致的缺陷，如 FC 帧封装、模拟点对点链路和访问常见服务。FCoE 帧是一种将 FC 帧作为整体进行封装而不做任何改变的以太网帧。保持 FC 帧的完整性能够简化 FCoE 和现有 FC SAN 的集成。如图 6.26 所示，封装后的 FC 帧具有相对较低的开销，只是增加了用于标记帧开始和结束以及用于填充以满足最小以太网帧尺寸要求的分隔符。FCoE 帧可以通过特殊的 Ethertype（以太网类型）值来与其他类型的帧进行区分。Ethertype 是以太网帧中的一个两字节报头字段，用于表示有效载荷的性质。Ethertype 值决定了接收端对接收帧进行何种处理。IPv4 和 IPv6 是普通的 Ethertype 值。为避免冲突，Ethertype 值是由 IEEE 注册管理机构进行管理的。

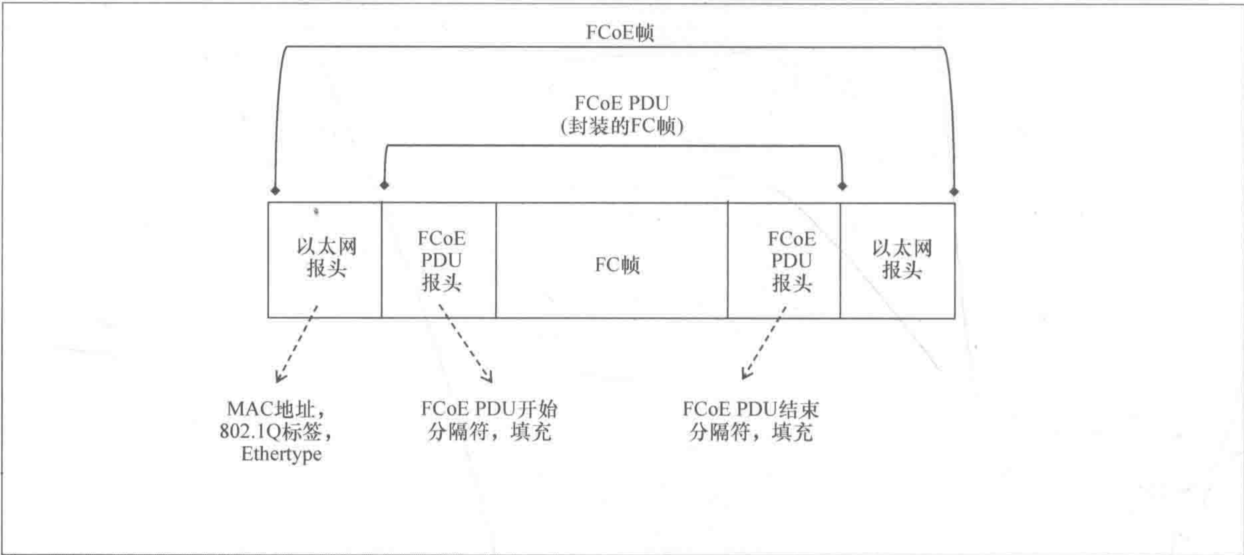


图 6.26　FCoE 帧结构

PDU—协议数据单元

可以将 FCoE 感知实体分为 FCoE 节点（FCoE Node，ENode）和 FCoE 转发器（FCoE Forwarder，FCF）。前者相对简单，实际上被称为将 FC HBA 和以太网 NIC（Network Interface Card，网络接口卡）合为一体的融合网络适配器。以太网 NIC 本质上是以太网中的 FC 交换机，它支持节点和 FC 交换功能，其中心任务是转发 FCoE 帧。对于接收到的每一帧，FCoE 转发器均对 FC 帧进行解封装，并根据 FC 帧

[1] 参见 IEEE 802.3 - 2008 标准附件 31B。

[2] 参见 ANSI INCITS 462 - 2010。该规范还定义了用于支持不同类型骨干网上的光纤信道的机制，包括 TCP/IP SONET/SONET（Synchronous Optical Network，同步光纤网络）/SDH（Synchronous Digital Hierarchy，同步数字系列）/OTN（Optical Transport Network，光传输网）/PDH（Plesiochronous Digital Hierarchy，准同步数字系列）和 MPLS（Multi - Protocol Label Switching，多协议标签交换）。



中的目标地址来确定转发 MAC 地址。然后，FCoE 转发器必须以适合以太网端口转发的方式再次对 FC 帧进行封装，将以太网源地址设置为 FCoE 转发器 MAC 地址，而将以太网目的地址设置为转发 MAC 地址。如果 FCF 还支持本地 FC 端口（因而是 SAN 网关功能），则可能不需要重新进行封装。FCoE 转发器可以将 FC SAN 绑定的 FC 帧转发到本地 FC 端口。

为了在以太网共享介质上模拟点对点链路，FCoE 需要依赖于虚拟端口和链路的逻辑结构。

虚拟端口模拟 FC 端口。这些端口通常是在 FCoE 节点和 FCoE 转发器上动态创建的。每个虚拟端口均与某个元素［称为 FCoE 链路终点（FCoE_LEP）］相关联，该元素用于处理封装和解封装，并处理以太网传输问题。两个虚拟点可以通过至少由相关 FCoE_LEP 的 MAC 地址标识的虚拟链路进行互连。该链路可用作在以太网网络上两个 MAC 地址之间传输封装 FC 帧的隧道。图 6.27 给出了 FCoE 概念性网络中的虚拟端口和链路。为了简单起见，没有显示 FCoE_LEP。由图 6.27 可见：

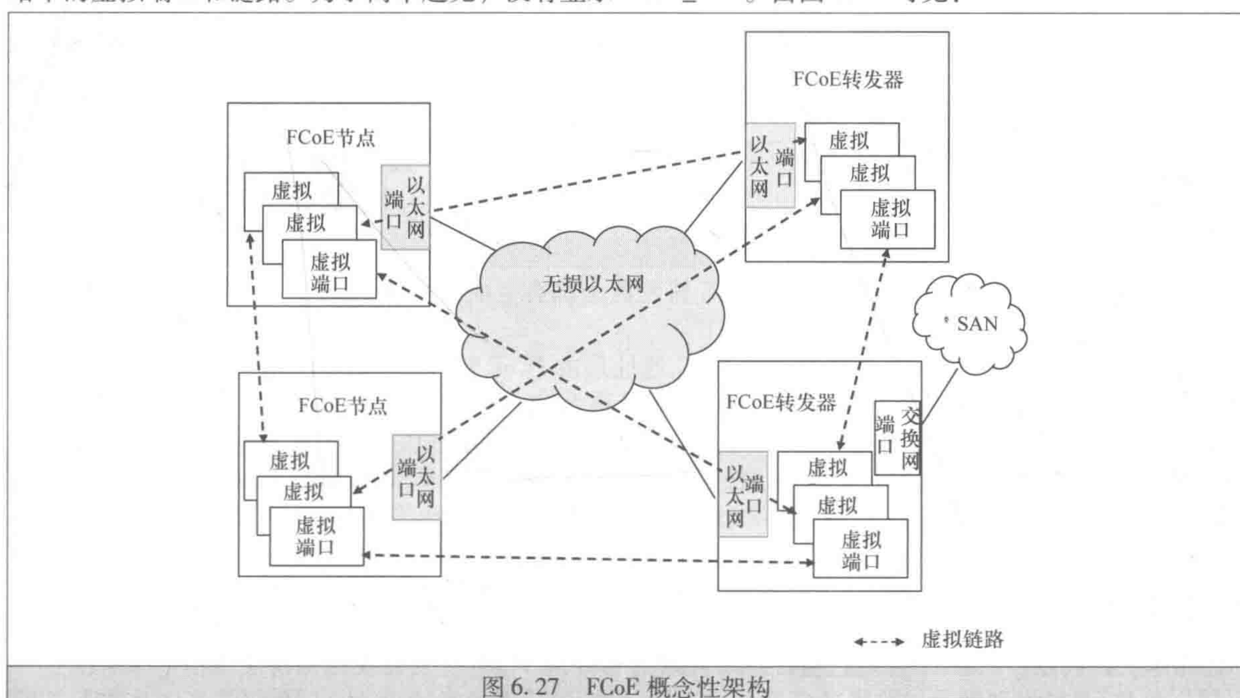


图 6.27　FCoE 概念性架构

1）FCoE 节点可以通过单个以太网端口与不同的 FCoE 转发器建立多条虚拟链路。

2）FCoE 转发器可以通过单个以太网端口与不同的 FCoE 节点建立多条虚拟链路。

3）FCoE 转发器可以通过单个以太网端口与不同的 FCoE 转发器建立多条虚拟链路。

4）FCoE 节点可以通过单个以太网端口与不同的 FCoE 节点建立多条虚拟链路。

现在，可以解释如何发现 FCoE 实体和虚拟端口，以及如何建立虚拟链路了。

在光纤信道（FC）中，终端设备与交换机之间配置直接物理链路。例如，为了执行交换网登录，设备只需通过链路向对应的已知端口地址发送请求。在 FCoE 中，FCoE 节点和 FCoE 转发器之间不存在直接物理链路。相反，存在着中间以太网链路和交换机。为执行交换网登录，末端节点必须首先建立一条适当的虚拟链路。如果这是手动完成的，则该过程效率低下且容易出错。

因此，FCoE 初始化协议（FCoE Initialization Protocol，FIP）应运而生。以太网帧中包含有 FIP 消息。特殊的 Ethertype 值能够将这些帧与 FCoE 帧区分开来。

FCoE 初始化协议涉及 FCoE 实体发现、虚拟链路实例化和虚拟链路维护。图 6.28 显示了 FCoE 节点和 FCoE 转发器之间的交互（两个 FCoE 节点或两个 FCoE 转发器的交互与其类似）。实体发现过程通常依托 FCoE 转发器将组播发现通告定期发送到已知组播地址。

FCoE 节点根据通告选择一种兼容的 FCoE 转发器，并在能力协商启动时发送一条发现请求。收到请求后，FCoE 转发器使用请求发现通告来响应 FCoE 节点，来确认协商后的能力。

一旦收到请求发现通告，FCoE 节点可以继续建立到 FCoE 转发器的虚拟链路。该过程类似于光纤信道中的交换网登录过程。

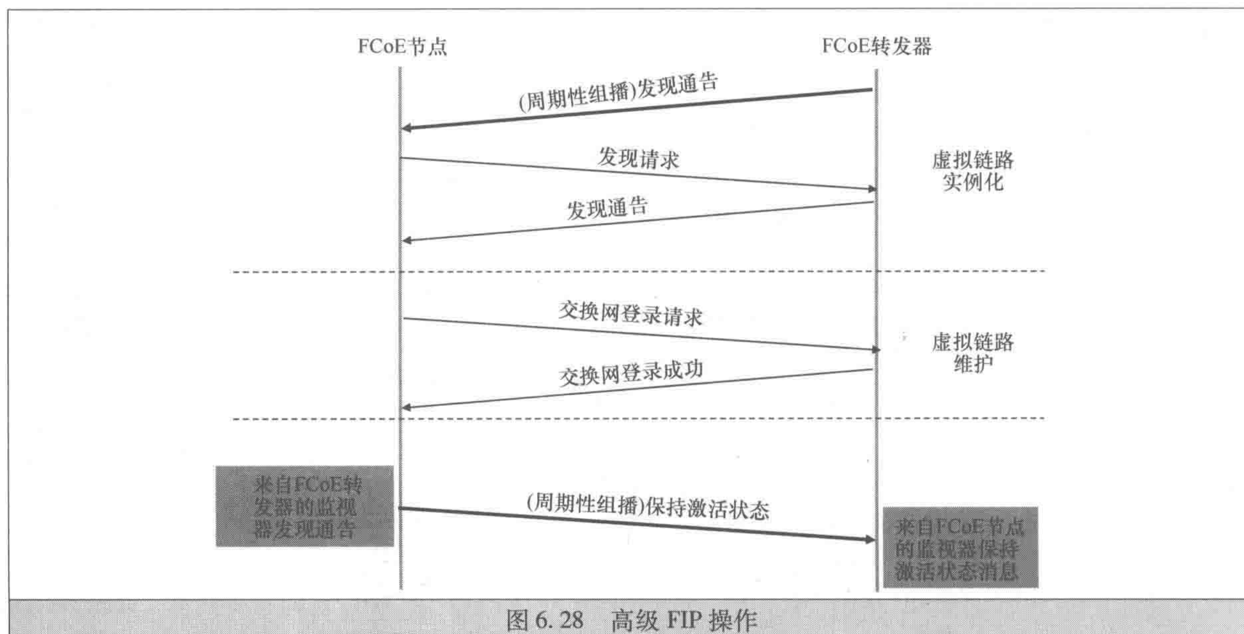


图 6.28　高级 FIP 操作

登录过程圆满完成后，FCoE 节点和 FCoE 转发器上都会生成一个虚拟端口，且 FCoE 节点和 FCoE 转发器之间会生成一条虚拟链路。

通常情况下，FCoE 节点上虚拟端口的 MAC 地址是由 FCoE 转发器分配的，虽然 MAC 地址也可以由 FCoE 节点分配。在前一种情况下，将 MAC 地址称为交换网提供的 MAC 地址（Fabric - Providing MAC Address，FPMA）。它是通过将 FCoE 转发器的 24 位 MAC 前缀和由 FCoE 转发器分配的虚拟端口的 24 位地址标识符进行级联得到的。这种方法可以确保交换网内的 MAC 地址是唯一的。虚拟端口和链路可以通过注销过程显式删除。该过程成功完成后，则可以释放所有相关资源，包括 MAC 地址和虚拟端口地址标识符。

虚拟链路可能会跨一系列以太网链路和交换机。因此，对于相关 FCoE 实体而言，因中间链路或交换机故障而导致的链路断开可能并非一目了然。FCoE 初始化协议通过使相关实体定期检查虚拟链路的状态来解决该问题。FCoE 节点通过监视组播发现通告并向 FCoE 转发器发送保持激活消息来解决这一问题，而 FCoE 转发器则通过监视来自 FCoE 节点的保持激活消息来解决这一问题（除了正在进行的发布组播发现通告任务之外）。

如果 FCoE 节点记录到两条通告丢失，则认为 FCoE 转发器上的虚拟端口是不可达的。在这种情况下，FCoE 节点将删除相关虚拟端口和链路。类似地，如果 FCoE 转发器记录到两条保持激活的消息丢失，则认为 FCoE 节点上的虚拟端口是不可达的。在这种情况下，FCoE 转发器将删除相关虚拟端口和链路。

为了更好地进行测量，FCoE 服务一般通过 VLAN（Virtual Local Area Network，虚拟局域网）提供，以便存储的流量被正确隔离。这些 VLAN 可能是预先配置的，但如果不是这样，则需要一种能够发现它们的机制。为此，FIP 包括一个在任何事件之前 FCoE 实体可以执行的附加过程。VLAN 发现过程非常简单。FCoE 节点（或 FCoE 转发器）将 VLAN 发现消息发送到预设的组播地址上。接收消息的 FCoE 转发器使用 FCoE 支持的 VLAN 标识符列表来响应 FCoE 节点。

iSCSI 是由伴随互联网发展而产生的泛在连通性支持的一种开发活动。如图 6.25 所示，这里某些对 SCSI 操作至关重要的功能（如可靠的依次传送、未确认数据包的自动重传和拥塞控制）用到了 TCP（Transfer Control Protocol，传输控制协议）㊀。

最初，存在着一种对潜在性能问题的担忧，但 TCP 的选择于 20 世纪 90 年代被南加州大学信息科学研究所在虚拟互联网 SCSI 适配器（VISA）项目[21]中进行了验证。该项目表明，TCP/IP 开销并不像

㊀ 在支持 SCSI 操作所必需的特征方面，流控制传输协议（Stream Control Transmission Protocol，SCTP）与 TCP 类似。然而，在 iSCSI 进行标准化时，通常 SCTP 认为太新、不可靠。

所担心的那么大，且可以通过使用功能更为强大的处理器进行补偿。随后，IETF 对相关标准化工作展开了讨论，并于2000 年第四季度成立了 IP 存储工作组。这一努力促使了一系列 RFC（Request For Comments，请求注解）的出版，从 2004 年详述 iSCSI 协议的核心 RFC 开始。位于以色列海法市的 IBM 研究实验室的 K. Meth 和 J. Satron 撰写的论文[22]对 iSCSI 协议的设计做了很好的解释。

为了解释 iSCSI 的工作原理，先来回顾一下图 6. 29 中的概念模型。这里的中心结构是代表 iSCSI 通信端点（启动器或目标）的 iSCSI 节点。该节点可以通过一个或多个网络门户从 IP 网络进行访问。

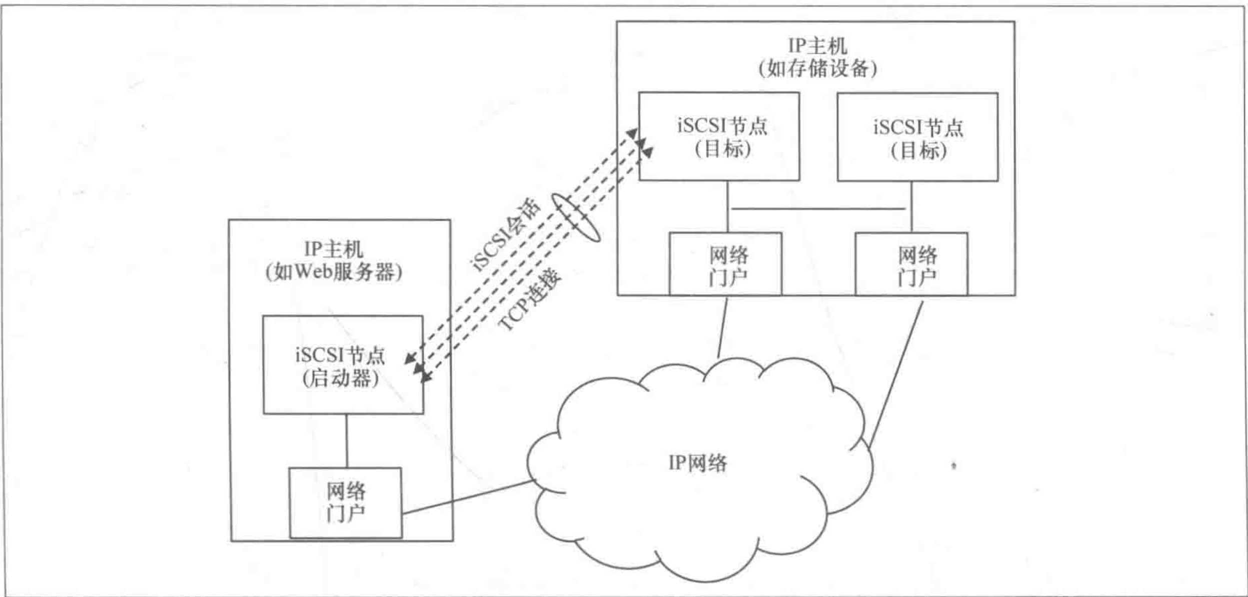


图 6. 29 iSCSI 概念模型

节点由全局唯一的 iSCSI 名称来标识，而 iSCSI 名称既不依赖于节点位置，又不依赖于 IP 地址。在同一地址处，可能存在着多个可达的 iSCSI 节点，且可以在多个地址处到达同一 iSCSI 节点。因此，在一对 iSCSI 节点之间的通信会话中使用多个 TCP 连接，以实现更高的吞吐量，这是完全可行的。

稍后会回到这一重要问题。图 6. 30 给出了针对 iSCSI 名称定义的两种格式：iSCSI 限定名（iqn）和可扩展唯一标识符（eui）。使用 iqn 格式，任何拥有域名的组织都可以发布这些名称。相比之下，eui 格式中的名称由 IEEE 注册中心（Registration Authority，RA）分配。

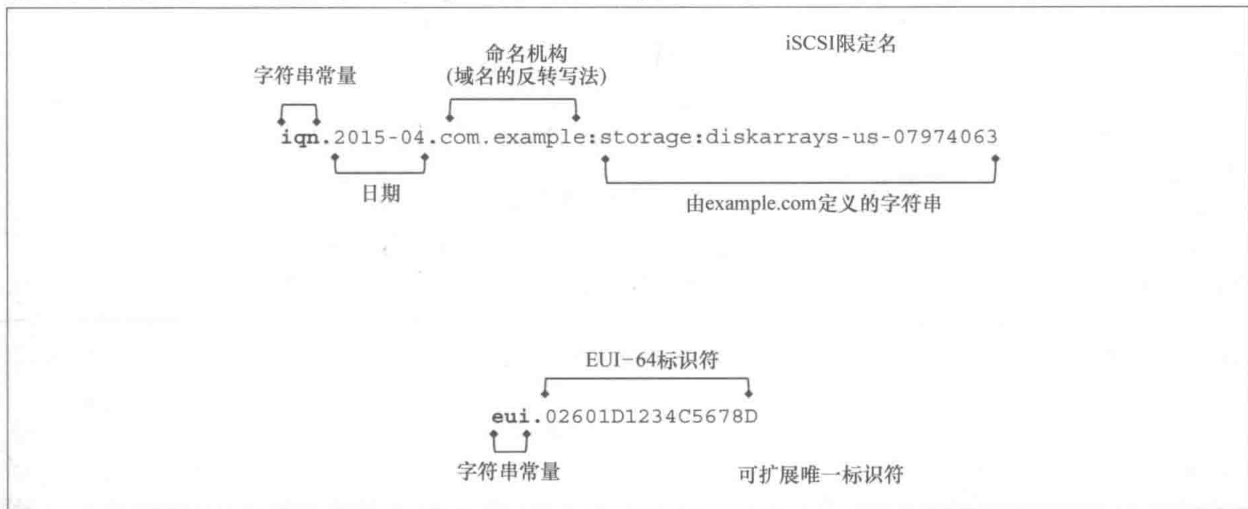


图 6. 30 iSCSI 名称

图 6. 31 描述了 iSCSI PDU（Protocol Data Unit，协议数据单元）的格式。只有基本报头片段域是必需的。此域包含有诸如 iSCSI PDU 类型和 SCSI CDB 等关键信息。

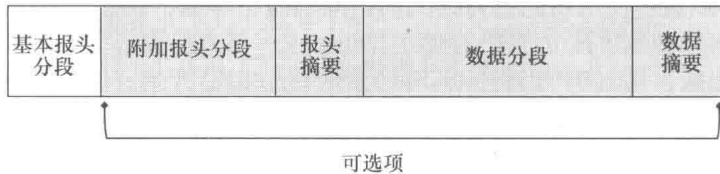


图 6.31　iSCSI 协议数据单元的格式

带有标签摘要的两个可选域包含用于检测报头和数据变化（如由噪声引起的变化）的校验和。为了降低处理成本，基本报头长度是固定的（48 字节），以容纳标准的 SCSI CDB。在 CDB 较大的情况下，需要用到附加报头分段。

iSCSI PDU 类型标识了 PDU 的关键功能，且人们已定义了若干种 PDU 类型。当然，一些 PDU 类型在 SCSI 中有直接对应物，如 SCSI 命令、SCSI 响应、SCSI 数据输入和 SCSI 数据输出。人们引入那些在 SCSI 中没有直接对应物的 PDU 类型，来为底层的 TCP/IP 提供必要的适配功能。这些 PDU 类型包括用于支持连接管理和能力协商的登录请求、登录响应、注销请求和注销响应，以及用于支持目标驱动流量控制的准备传输（Ready to Transfer，R2T）。

为了说明不同类型 iSCSI PDU 的作用，图 6.32 描述了一个写操作的信息流实例。

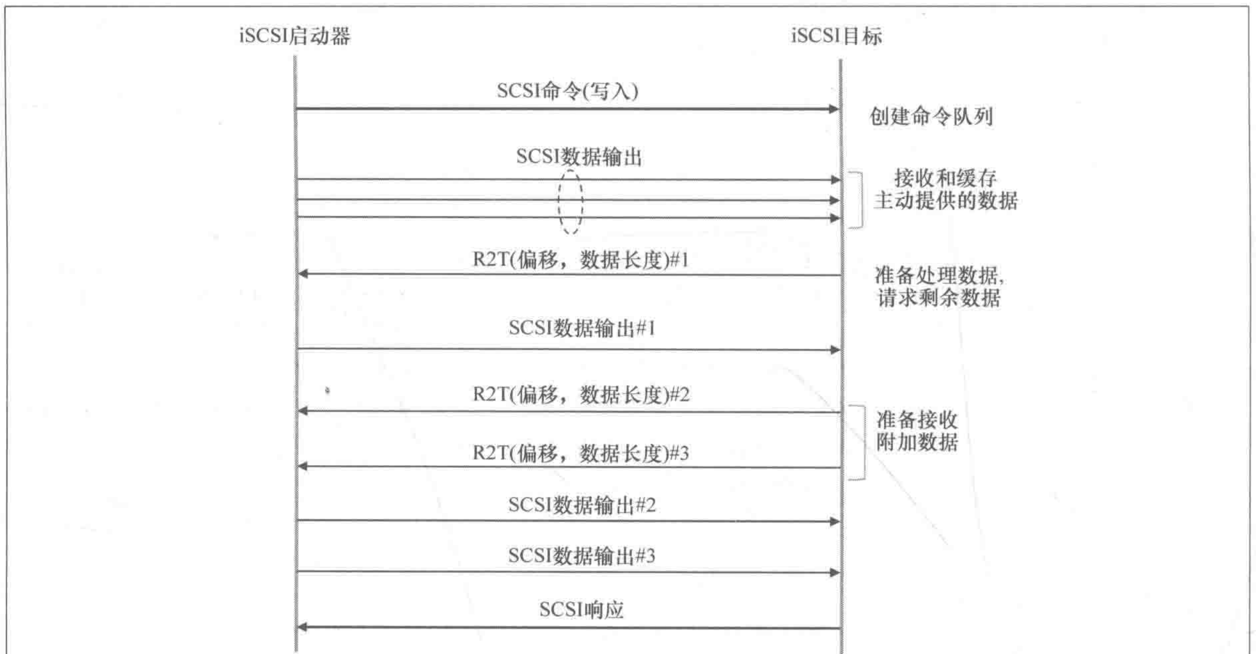


图 6.32　写操作的工作流程

每个 SCSI 命令协议数据单元（PDU）必须拥有一个匹配响应 PDU，用于表明命令是否成功执行。在发出 SCSI 响应之前，可能需要在启动器和目标之间进行数据传输。此传输由 SCSI 数据输入和 SCSI 数据输出 PDU 负责执行。在该实例中，启动器在写入命令之后，将预期数据发送给几个 SCSI 数据输出 PDU 中的目标，直到达到预先协商的非请求数据上限为止[⊖]。随后，启动者仅在目标请求时发送任何其他内容。

R2T PDU 通知启动器哪部分数据需要发送。目标直接发送 R2T PDU，而不需要等待针对旧 R2T PDU 的响应。在接收到 R2T PDU 之后，启动器发送 SCSI 数据输出 PDU 中所请求的数据。目标驱动方案支持基于目标负载和配置进行局部优化。但是这种方案是以传输额外 R2T PDU 为代价的，当数据量小、网络时延长时，这种方案可能会变得不可接受。这就是为什么 iSCSI 支持启动器在未请求的情况下将数据传输到目标，正如前面流程中所描述的那样。也可以采用另一种称为实时数据的更为高效的方

⊖ 在读操作的情形中，目标在收到读取命令之后，将所请求的数据发送给一个或多个 SCSI 数据输入 PDU 中的启动器。

案，它支持数据作为 SCSI 命令 PDU 的一部分进行发送。在启动器和目标之间建立 TCP 连接之后发生的登录过程中，需要对非请求数据的最大数量进行协商。稍后将讨论登录过程。

使用 TCP/IP 作为传输协议存在的一个问题是底层物理介质未得到充分利用。作为补救措施，引入了 iSCSI 会话的概念。iSCSI 会话是一组用于在启动器和目标建立链路的 TCP 连接。TCP 连接数可能会随着时间的推移而增加和减小，从而支持聚合多个 TCP 连接以实现更高的吞吐量。

当多个连接可用时，会出现在执行 I/O 的上下文中正确使用多个连接的问题。当然，使用单独连接进行控制和数据传输以确保连接始终可用于任务管理是合理的。然而，这种方案需要跨多个连接进行监视和协调，甚至需要在启动器或目标上安装不同的适配器。

为了避免这种复杂性，iSCSI 采用了一种称为连接效用的方案。采用此方案，启动器可以使用任何连接来发布命令，但必须坚持使用同一连接来完成所有后续通信。

需要对 iSCSI 会话进行管理。会话管理的很大一部分是由 iSCSI 登录过程来处理的。登录过程的成功完成会导致新会话或添加连接到现有会话中。

该过程的先决条件是启动器要知道所用存储设备（即目标）的名称和地址。一种方法是在启动器中预先配置此类信息。然后，发生任何改变都需要重新进行配置。

另一种方法是基于服务定位协议。它支持启动器动态发现可用目标。为启动登录过程，启动器首先建立一条到目标上已知 TCP 端口[⊖]的连接。一旦连接建立，启动器将通过登录请求和登录响应 PDU 来执行登录步骤。相互认证可以通过协商的认证方法进行，并将挑战－握手认证协议（Challenge Handshake Authentication Protocol，CHAP）作为默认认证方法。至少操作参数是可协商的，其中包括非请求数据的最大数量、SCSI 数据输入 PDU 的最大尺寸，以及在 PDU 中是否包含循环冗余校验和以进行误差保护。当一切顺利时，目标发送一个登录响应 PDU，表示登录过程已成功完成。只有这样，与会话关联的新连接才能用于 SCSI 通信。

跨多个 TCP 连接的有效负载分布和错误恢复也是会话管理的一部分。iSCSI 支持用于错误恢复的 3 级层次结构，其复杂性明显增加。

底层是会话恢复，它再次重构一次无效会话。这涉及清理所有相关工件（如关闭所有 TCP 连接并中止所有具备错误指示功能的、即将执行的 SCSI 命令），然后重新建立一组新的 TCP 连接。

第二层是摘要故障恢复，除了会话恢复之外，该层支持具有不匹配数据摘要的 PDU 接收器请求重新发送 PDU。

最后，连接恢复包括摘要故障恢复，且还支持将断开连接上即将执行的命令传输到另一条连接（可能需要创建）上。

每种恢复过程仅适用于特定环境。例如，在局域网（LAN）中，出现任意类型的错误都比较罕见，因而只需要进行会话恢复就足够了。总而言之，只要在启动器和目标之间有可能建立连接，则 iSCSI 会话即可保持激活状态。当最后一条连接关闭时，会话终止。要使多个连接在启动器和目标之间显示为单个 SCSI 互连，iSCSI 需要使用序列号和标签。

在 iSCSI 中，会话标识符由启动器部分和目标部分构成。前者（启动器会话标识符）由启动器在会话建立时显式分配；后者由 TCP 端点选择的启动器隐式表达。为了确保启动器与给定目标（特别是当启动器分布式部署时）进行的每次会话标识符唯一，需要指定一种由注册机构控制的分层命名空间。

必须强调，可能是登录过程组成部分的相互认证步骤只是一次性事件。对于后续通信是否仍然发生在认证节点之间没有任何影响。此外，iSCSI 本身不提供任何保护连接或会话的机制。所有本机 iSCSI 通信都使用明文，易受到窃听和主动攻击。在不受信任的环境中，iSCSI 应与 IPSec 一起使用。

6.2.5　对象存储 ★★★

网络连接存储（NAS）以独立于操作系统的方式提供对共享数据的文件级受控访问。其设计要求来自于客户端的每个与文件相关的 I/O 请求通过文件服务器，该服务器充当文件存储设备的适配器。因此，文件服务器是限制 I/O 吞吐量的潜在瓶颈。解决这一限制的方法之一是允许客户端通过共享元数据直接访问存储设备[23]。然而，实现的性能增益却是以牺牲安全性为代价的。传统的块存储设备相

⊖ 由 IANA（Internet Assigned Numbers Authority，互联网号码分配机构）分配的 iSCSI 知名 TCP 端口号是 3260。

对简单，它可以读取和写入0、1数据块，但不具备理解数据含义或辨别诸如文件或目录等结构的能力。访问控制只适用于整台设备，可以赋予客户端访问所有内容的权利，也可以禁止其访问任何内容。这显然是有问题的。此时，对象存储应运而生。

一种相对较新的技术（称为对象存储）支持跨多种操作系统并以直接存储访问的速度安全地进行数据共享。它的主要特征是：①比块级更高的抽象级新型设备接口；②设备本身的附加智能。通过新接口（INCITS T10技术委员会已经完成了标准化工作），存储设备作为对象集合出现。对象是平面命名空间中唯一可识别的有序字节集。对象可以包含任何类型的数据，无论是文件、数据库，还是整个文件系统。对象的组成部分是什么取决于存储应用。当对象被创建、删除、修改或复制时，分配和释放块的相关任务由存储设备进行执行。为了跟踪在用块和空闲块，设备依赖于每个对象的元数据。

设备附加智能是指理解元数据、管理空间和支持细粒度访问控制等功能。这种设备内置的功能支持新性能优化机制（如文件预取和数据重组），简化应用聚类，并支持自动化存储管理。

细粒度访问控制是云计算的基础。ANSI INCITS 458－2011[㊀]标准中定义的访问控制机制是建立在能力和证书上的概念。该能力描述客户端访问对象的权限，如读取、写入、创建或删除。本质上，证书是一种使用加密保护的防篡改能力，包括使用共享密钥生成密钥散列消息认证码（Hash Message Authentication Code，HMAC）的能力[㊁]。更具体地说，证书是一种结构：

〈能力，对象存储标识符，能力密钥〉

其中

能力密钥 = HMAC（密钥，能力 ‖ 对象存储标识符）

图6.33描绘了概念模型。安全管理器负责根据客户端请求的策略授予证书。用于计算能力密钥的秘密密钥在安全管理器和对象存储设备之间进行共享。当且仅当客户端提供它拥有合法证书的证明时，对象存储设备才将执行命令。因此，存储设备充当的是访问策略实施者的角色。

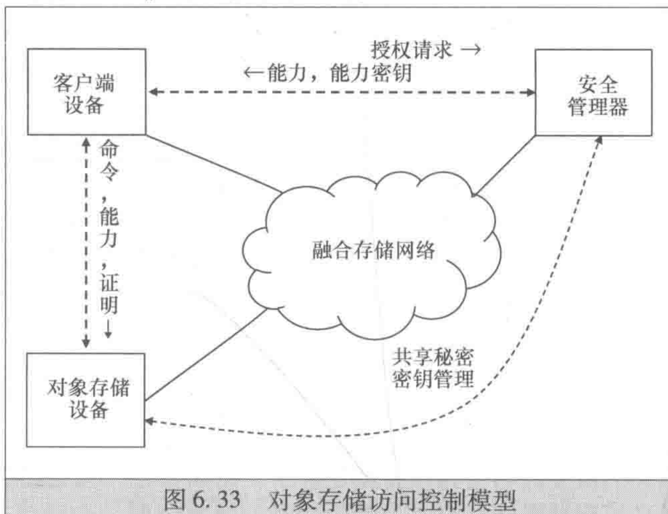


图6.33　对象存储访问控制模型

现在的问题是如何提供合理的证明。就像驾照证明司机具备驾驶某类车辆资格一样，证书本身可以作为证明吗？

为了回答这一问题，考虑一下相关证明的基本要求。至少来说，证明应该是可验证的、防篡改的、难以伪造的，且能防止未授权使用。证书需要满足除最后一条之外的所有要求；没有内置机制将其绑定到获取客户端或客户端和存储设备之间的通信信道上。相比之下，驾照有一张司机照片，能够将驾照与驾驶员绑定，虽然手上的问题并不需要这么强大的约束力。显然，这并不理想，特别是在未得到有效保护的存储传输中证书易受到窃听的情况下。

因此，另一种证明方案是可用的。标准化方案基于能力密钥来得出证明。证明是根据协商的安全方法通过选择性请求组件的能力密钥来进行量化计算。下面的安全方法是可行的：

1）NOSEC。也就是说，不使用访问控制。在这种情形中，存储设备执行命令而不需要任何证明。此方法仅适用于完全隔离的环境。在这种环境中，链路是安全的，且客户端和存储设备之间存在着信任关系。

2）CAPKEY。在这种情形中，证明是客户端和存储设备之间信道标识符的完整性检查值。因此，正在使用的特定频道是固定的（信道标识符由对象存储设备分配，客户端可从该对象存储设备处获取信息）。该方案假设信道本身是安全的。它可以防止证书在不同信道上未经授权使用，同时支持授权，

㊀ ANSI INCITS 458－2011，基于对象的存储设备命令－2（OSD－2）。

㊁ 参见FIPS 198（2002），密钥散列消息认证码（HMAC）和FIPS 180－1（1995），安全散列标准。

即把证书转发给另一个客户端[24]。对于给定的信道，该方案相当简单。客户端不需要请求证书或单独计算每条命令的完整性检查值。

3）CMDRSP。在这种情形中，证明是请求中命令的完整性检查值。该方案适用于客户端和存储设备之间信道不安全的环境，但为用户数据提供完整性保护是不切实际的。除了命令源认证和完整性保护外，它还通过在每条请求中使用随机数（nonce㊀）来提供抗重放保护。Factor 等人撰写的论文[25]解释了相应的随机数管理机制。

4）ALLDATA。在这种情形中，该证明包含多个完整性检查值（包括 CMDRSP 的完整性检查值）。额外的完整性检查值分别是通过对存储设备发送和接收的数据进行计算得到的。因此，在 CMDRSP 提供的功能之上，该方案能够为客户端和存储设备之间交换的数据提供抗重放和防篡改保护。对于 CMDRSP 来说，它适用于通信信道不安全的环境。

需要注意的是，访问控制机制不涉及实际的客户端认证。由此导致的客户端和存储设备解耦能够提高可扩展性，支持存储设备可以独立于特定客户端进行扩展。然而，当流氓客户端可以访问存储设备时，需要一种撤销证书的方法。这里有两种解决方案。

一种方案是安全管理器和对象存储设备更改相关的秘密密钥。这种方案相对易于实现，但它存在单个对象之外的系统效应。一旦更改密钥，所有未完成的证书均将失效。

另一种方案是安全管理器重置存储设备中有问题对象的策略访问标签。标签也是能力结构的一部分。合法证书必须具有与设备存储内容相匹配的策略访问标签。

6.2.6 存储虚拟化 ★★★

存储虚拟化[26]涉及物理存储的抽象，来向应用屏蔽其底层细节（如实际介质、访问接口和位置㊁）。因此，物理上分散的异构存储系统能以单个聚合实体的形式出现，反之亦然。例如，对于操作系统来说，10 个 800GB 的硬盘可以模拟为 1 个 8TB 的虚拟硬盘。相反，可以将 1 个 8TB 的硬盘分区为 8 个 1TB 的虚拟磁盘，并将其分配给不同主机。逻辑上分配存储空间的灵活性还支持动态资源管理，进而提高整体资源的利用率。

通常，存储虚拟化需要：①管理将逻辑存储映射到物理设备的元数据；②根据映射来转换和重定向 I/O 操作。文件级虚拟化建立在块级虚拟化之上。文件级虚拟化要求将存储卷以文件呈现。如图 6.34 所示，使用虚拟化实体，多个文件服务器可以单个虚拟文件服务器的形式出现。

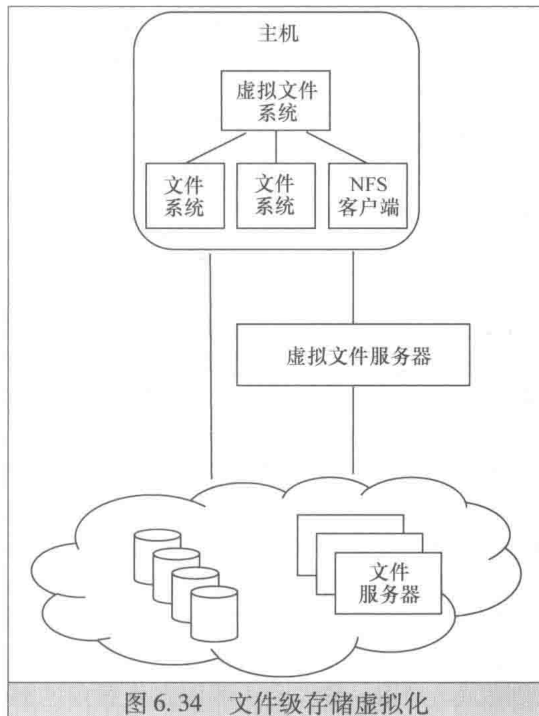


图 6.34 文件级存储虚拟化

通过使用块级存储虚拟化，对操作系统来说，存储可以一组逻辑卷或虚拟磁盘的形式呈现（逻辑卷可以组合不连续物理分区并跨多台物理存储设备）。

如图 6.34 所示，根据虚拟化完成的位置（主机、网络或存储设备），存在着 3 种块级虚拟化方法。在基于主机的方法中，虚拟化由卷管理器进行处理，且卷管理器本身可能是操作系统的一部分。卷管理器负责将本地块映射到逻辑卷，同时跟踪整体存储利用率。在理想情况下，映射应提供动态调整功能，即根据特定应用的最新需求来增大或减小虚拟存储容量。该方法的主要缺点是逐个主机控制不利于提高多主机环境中的最优存储利用率，更不用说卷管理器的操作开销成倍上升。

㊀ 单词 nonce 是一个只被使用一次的字符串（作为通信交换的一部分）。这是一种确保新鲜度的机制。以支票簿为例，里面的每张支票都应该是唯一的（这样可以非常容易地将副本与原件区分开来），且为了确保唯一性，人们使用了包括序列号和水印在内的诸多机制。

㊁ 从这个意义上讲，熟悉的文件系统结构其实是一种存储虚拟化形式。

在基于存储设备的方法中，虚拟化由存储系统的控制器进行处理。由于控制器与物理存储器距离非常近，因而这种方法往往能够达到良好性能。然而，它具有过分依赖供应商和难以（即便有可能）跨异构存储系统工作的缺点。

在基于网络的方法中，虚拟化由存储网络中的特殊功能实体来处理，且该实体可能是交换机的一部分。只要主机和存储系统支持适当的存储网络协议（如 FC、FCoE 或 iSCSI），该方法对它们就是透明的。根据控制流量和应用流量的处理方式，可以进一步将其分为带内（对称）或带外（不对称）方法。

图 6.35 对带内方法进行了描述，其中用于映射和 I/O 重定向的虚拟化功能实体始终位于控制和应用流量的路径中。当然，虚拟化功能实体可能会成为瓶颈和单点故障。缓存和聚类是缓解这些问题的常用技术。

从积极的方面看，由带内方法提供的中心控制点简化了管理，并支持诸如快照、复制和迁移等高级存储功能。快照功能与云计算密切相关。它可以用于捕获虚拟机在某个时间点的状态，反映其组件（如存储器、磁盘和网络接口卡）的运行状况。状态信息允许在应用补丁或故障后回滚。然而，此时存在着一种折中，因为在这种情况下，在创建虚拟机的快照时，同一主机上其他虚拟机的性能可能会受到影响。

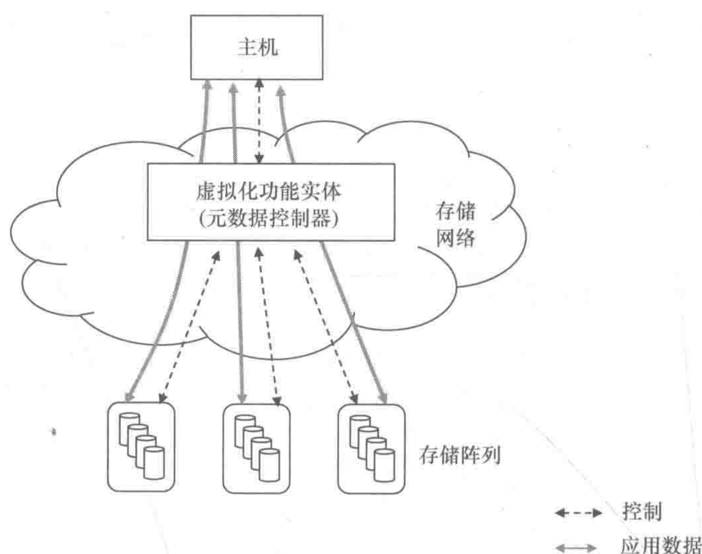


图 6.35 带内存储虚拟化

图 6.36 对带外方式进行了描述，其中虚拟化功能实体位于控制流量的路径上，但没有在应用流量的路径上。虚拟化功能实体可以为应用流量定向。与带内方法相比，该方法具有更好的性能，因为应用流量可以直接进入目的地，而不会在虚拟化功能实体中导致任何处理延迟。但这种方法并不支持高级存储功能。更重要的是，它要求主机对控制和应用流量加以区分，并准确地路由流量。因此，主机需要增加虚拟化适配器，它还可能会支持缓存元数据和应用数据以提高性能。然而，每个主机的缓存面临着保持分布式缓存一致的挑战性问题。

鉴于相对透明度和存储池的灵活性，基于网络的方法最适用于云计算。采用这种方法，可以将存储分配给 VM（虚拟机）主机，而 VM 主机又可以通过本章参考文献［27］中描述的自身虚拟化设施将分配到的虚拟存储分配给虚拟机。然而，带内和带外方法之间的选择并不十分清晰，主要取决于应用。理想情况是采用一种综合了两种方法优点的混合方法。显然，使用智能交换机实际上是可以实现这一点的，它主要负责处理带外控制流量和带内应用流量[28]。

6.2.7 固态存储 ★★★

存储技术在性能、成本和其他属性方面差别很大。表 6.3 对存储技术差异进行了汇总。其中给出的访问时间和成本与磁盘的访问时间和成本相关。磁盘的访问时间为毫秒级。静态随机存取存储器



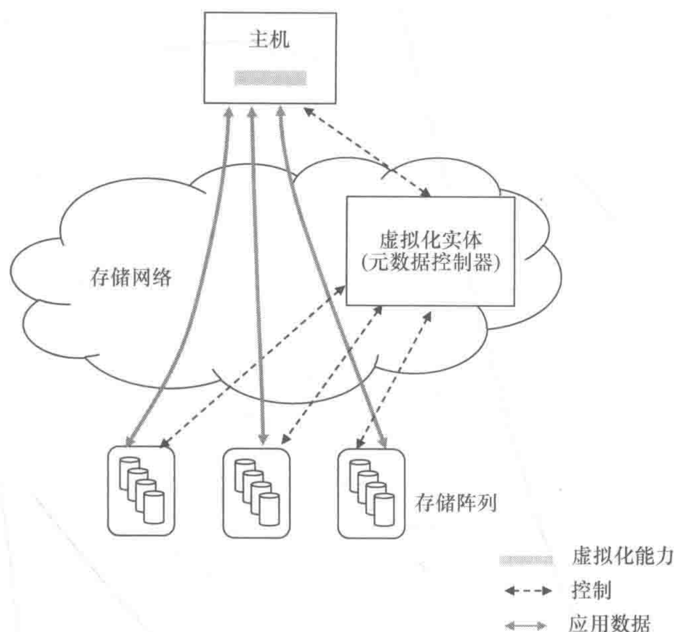


图 6.36　带外存储虚拟化

表 6.3　存储技术的比较

	相对访问时间	相对成本	保持时间
SRAM	$\sim 10^{-7}$	$\sim 10^{4}$	接通电源的持续时间
DRAM	$\sim 10^{-5}$	$\sim 10^{2}$	$\ll 1s$
闪存	$\sim 10^{-2}$	~10	若干年
磁盘	1	1	若干年
磁带	$\sim 10^{-4}$	0.5	若干年

注：SRAM—静态随机存取存储器；DRAM—动态随机存取存储器

(Static Random Access Memory，SRAM)、动态随机存取存储器 (Dynamic Random Access Memory，DRAM) 和闪存比硬盘快得多。

在没有移动部件和完全电子处理的情况下，这些存储器在其他方面（如抗冲击性和能量效率）也是非常优异的。在动态随机存取存储器的情况下，数据作为电荷存储在电容中。由于电荷随着时间而泄漏，因而电容需要定期进行刷新。定期刷新的需求解释了将此类 RAM 命名为动态的原因。使用静态随机存取存储器，数据是存储在晶体管而不是电容中的。静态随机存取存储器不需要进行刷新，这一点使得 SRAM 比 DRAM 快。然而，SRAM 和 DRAM 都是易失性的，因为它们需要电源来保持存储的数据。相比之下，闪存是非易失性的。

闪存是一种电可擦除可编程只读存储器（Electrically Erasable Programmable Read – Only Memory，EEPROM）[㊀]。将数据保存在浮栅岛中，它可以长时间（可达数年）保持电荷。20 世纪 80 年代，藤尾增冈博士在东芝工作时发明了一种相对较新的闪存技术。将该技术命名为闪存，因为它能够快速擦除大块内存。实际上，Fujio Masuoka 博士在 1984 年和 1987 年分别设计了两种闪存[㊁]。第一种类型称为 NOR 闪存，因为其基本结构具有类似于 NOR（或非）门的属性。NOR 闪存速度非常快（至少比硬盘快），且可以随机地寻址到给定字节。NOR 闪存的存储密度有限。

㊀ “电可擦除”意义重大，因为早期的可擦除可编程只读存储器（EPROM）技术需要暴露于紫外线（超过 10min）以擦除内容。可编程 EPROM 也需要单独的专用设备。

㊁ Fujio Masuoka 博士在接受采访时[31]指出，他在 1982 年发明了 NOR 闪存，但在 1984 年才发表了与其相关的首篇文章。因此，在大多数文献中，通常认为 NOR 闪存的诞生时间是 1984 年。

第二种类型的闪存消除了这一限制条件（同时也降低了成本）。称其为 NAND 闪存，因为其基本结构具有类似于 NAND 门的属性。然而，NAND 闪存仅允许以大于 1 字节的单位进行随机访问。NAND 闪存在消费类电子产品中占有一席之地[29]，且比 NOR 闪存应用范围更为广泛，如数字相机、便携式音乐播放器和智能手机等。

处理存储技术巨大差异的方法是实现分层存储。图 6.37 描述了存储器层次结构，其中存储器的层次越低，它与处理器的距离越近，速度越快，成本越高，且尺寸越小。不同层次存储器尺寸的约束条件意味着数据可以全部存储在较高层次上，该层的存储器具有足够大小。此外，保存在某层存储器的数据只能是保存在上层存储器的数据的一个子集㊀。总之，存储器层次结构旨在让用户产生一种无限快速存储器的错觉。将数据保存在存储器层次结构中的一般策略是将最近访问的数据项保存在处理器附近，且在复制数据项到较低层时将其邻近层包含进来。对于存储器层次结构的深入讨论，我们推荐大家阅读 John L. Hennessy 和 David A. Patterson 编著的计算机架构教材[30]。

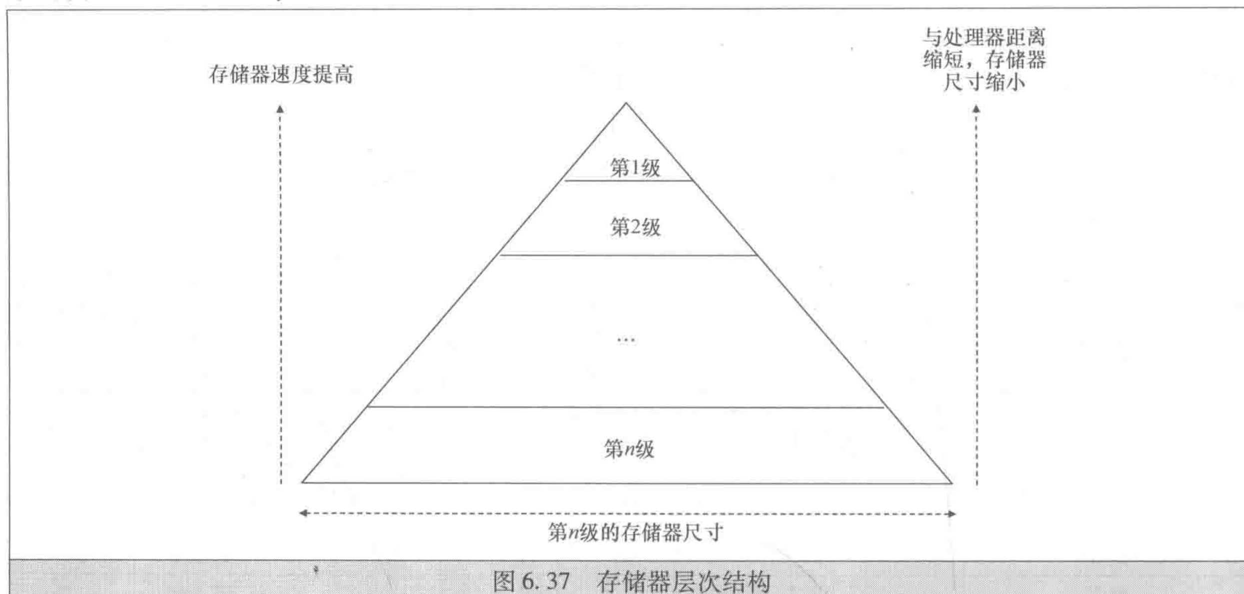


图 6.37 存储器层次结构

直到最近，存储器通用层次结构是：SRAM 用于缓存、DRAM 用于主存储器、硬盘用于分页存储器。固态存储技术的不断发展引入了其他实用的解决方案。

特别需要指出的是，NAND 闪存已经成为硬盘的第一个有力挑战者。它在存储器层次结构中越来越多地应用于 SRAM 和硬盘之间。当以这种方式使用时，闪存被塑造成固态驱动器，它能够模拟硬盘驱动器来协助实现与现有系统的集成。

在不久的将来，虽然固态硬盘的价格（每字节的成本）仍会比硬盘驱动器更加昂贵，但它们可以提供更好的 I/O 性能。固态硬盘驱动器在随机读取操作方面的性能要比硬盘高大约 3 个数量级。尤其适用于那些涉及高比例随机 I/O 操作的应用。Web 和在线事务处理服务是此类应用的常见实例。云服务是一个更好的例子，因为它们将不相关的工作负载复用在同一硬件上，即 I/O 混合器效应。

要在云中部署固态硬盘，必须克服 NAND 闪存固有的 3 个限制条件：

1）对现有内容进行写操作需要首先擦除此内容（这使写操作比读操作慢得多）。

2）擦除操作是以块为单位完成的，而写操作是以页㊁为单位进行的。

3）在经历过有限个写入 - 擦除周期之后，存储单元耗尽。

考虑到这些限制条件，直接更新已有页面内容将会导致高延迟，因为需要对整个块进行读取、擦除和重新编程。显然，这是不可取的，会导致重定位写入（或异地写入）的事实。这里，需要将最新数据写入一个空闲页面，而将旧页面标记为无效。

㊀ 在严格的内存层次结构中，信息只能从一层复制到相邻层。

㊁ 在块中，可以对页面进行组合。典型的块包含 32、64 或 128 页（它往往比文件系统中的块大）。页面尺寸是变化的。例如，1 页可以包含 512、2048 或 4096 字节。

在牺牲越来越多无效页面的情况下，写入性能得到了改善。如果不进行释放，则无效页面将会迅速耗尽存储空间。要释放存储空间，垃圾回收是必要的。

对于纯序列写入操作，垃圾回收非常简单。当逐页写入数据时，可以对块进行逐个报废和回收。这不需要额外的写入 - 擦除操作。相比之下，随机写入操作的情况要复杂得多，因而需要更加复杂的算法。例如，某种算法可以实现回收页面数量的最大化或额外读写操作数量的最小化。垃圾回收算法的有效性取决于它所导致的写入操作放大程度。

因为存在写入放大问题，所以在 NAND 闪存上执行的写入操作最终数要大于主机所要求的写入操作数。当然，由于性能和耐久性方面的原因，最好还是要尽可能遏制写入放大现象。这种遏制在云计算中尤其重要，因为随着写入操作量的增加，写入放大现象会越来越严重。

存在着两种用于遏制写入放大的技术。

一种技术是超量配置，即将用户地址空间限制为原始内存容量的一小部分。超量配置会增加选择用于回收的块中的无效页面数量。该技术是有效的，且不依赖于操作系统（或文件系统）的特殊支持。

该技术采用特殊的 ATA 命令来通知底层存储器需要删除哪些数据。使用硬盘时不需要这种命令，因为存储介质支持就地写入。在 NAND 闪存的情况下，该命令有很大的不同，因为它节省了垃圾收集期间浪费的时间。

为了延长 NAND 闪存的使用寿命，可采用“耗损均衡”的做法以尽可能均匀地扩展写入 - 擦除操作。需要注意的是，异地写入本质上支持耗损均衡。这种做法引入了写入放大。为了考察其效果，图 6.38 给出了极端实例。

在图 6.38 中，单元对应于页面，它们可能是有效的、无效的或空闲的（未标记的），而行对应于块。每个块的阴影表示其年龄。块的阴影越暗，块的年龄越大。块 5 是最老的，达到生命的终点，而块 1 是最年轻的，仍然拥有诸多写入 - 擦除周期。给定状态后，除非腾空另一个块（即空闲块）来替换块 5，否则内存也将不再可用。

块1	√	√	√	√	√	√	√	√	√	√
块2	√	√	×	√	√	√	×	×		
块3	√	√	√	×	×	√	×	×	×	×
块4	√	×	×	√	√	√	×			
块5										

图 6.38　NAND 闪存的假设状态

假设存储在块 1 中的数据是静态的，则它是可腾空的最佳块，原因有二：①一旦使用块 1 的数据进行编程，则不再需要对块 5 进行更新；②存储器的寿命延长了最多的周期。将块 1 的内容重定位到块 5 需要比任何其他解决方案更多的读取和写入操作。因此，耗损均衡不仅涉及未使用的周期，而且还涉及移动未变化数据浪费的周期。本章参考文献［32］对写入放大和耗损均衡进行了透彻的分析。

尽管云服务成本高昂，可是其独特性能仍然为基于 DRAM 的存储设备留足了空间。为了解决波动问题，这些设备配有内置电池或其他备用电源。更为有趣的是，斯坦福大学曾经研发过一个名为 RAMCloud 的项目[4]。该项目旨在创建一种在数据中心使用的新存储类型，它可以随时将所有数据保存在 DRAM 中。商用服务器是构成模块。根据所需规模，可以将数以千计的商用服务器集成为单个统一大型存储系统。预计这种存储系统具有卓越的性能，但同时也面临着若干个挑战。除了其他方面（如高度分布式存储和低时延网络的管理）的挑战外，存储在 RAMCloud 上的数据应该与存储在硬盘上的数据一样持久耐用，电源故障不能导致永久性数据丢失；单个存储服务器的故障不能导致数据丢失或数秒内不可用。这些需求就是复制和备份技术的需求，它们可以充分利用非易失性磁盘存储，同时保持由 DRAM 提供的原始性能优势。“缓存日志记录”[33]就是这样的技术。需要注意的是，缓存日志记录与前面讨论的文件系统日志记录有关。它同时使用磁盘和存储器进行备份。可以将主存储服务器上的数据变化以日志条目的形式同步复制到备份存储器上，但以异步方式复制到磁盘上。存储器副本是暂时的。对副本进行缓存，然后批量传输到磁盘。一旦复制到磁盘上，则将日志条目从备份存储器中删除。缓冲记录确保了性能，但会留下潜在的问题：如果主存储器和所有备份存储器同时掉电，则缓存数据将丢失。这一问题的直观解决方案是为每台存储服务器配备一枚小电池，以便能够在掉电后将缓存的

日志条目刷新到磁盘。颇具讽刺意味的是，这种特殊配置偏离了使用商用服务器的原始假设。

从本质上讲，RAMCloud 提供了近乎无限容量的远程缓存以提高应用性能。在这方面，它受到 Memcached[5] 的影响，Memcached 是 Brad Fitzpatrick 最早开发的开源分布式缓存系统，用以提高 LiveJournal 的性能[34]。Memcached 能够为商用计算机上 DRAM 中的小块任意数据提供简单的键值存储。它特别适用于缓存，且支持应用绕过诸如数据库查询之类的大量操作。数据持久性不是该解决方案重点考虑的一部分；每个缓存条目仅在一段时间内有效。Memcached 是基于客户端/服务器的，它采用的是请求/响应协议（可以通过采用 TCP 或 UDP 运行）。

服务器将数据存储在散列表中。键值是用于索引到表中的唯一字符串。例如，数据库查询结果可以缓存在将查询字符串作为键值的 Memcached 服务器中。尽管每个数据项的使用寿命都是非常有限的，但 Memcached 并没有通过垃圾回收来主动释放内存。相反，只有当检索到过期数据项或者当需要空间用于缓存新数据项时，才会考虑释放存储器。在后一种情况下，将丢弃最近最少使用的数据项㊀。如果过期的项目存在，则首先选择它来进行回收。否则，选择一个仍然有效的项目。

根据服务器上可用的 DRAM 大小，缓存工作负载数据可能需要多个服务器。在这种情况下，散列表分布在多个服务器上，这些服务器形成具有聚合 DRAM 的集群。在设计时，Memcached 服务器相互既不了解又不能统一协调。客户端的工作是选择使用什么样的服务器，且客户端（拥有在用服务器的信息）基于要缓存的数据项键值来完成这一工作。

如何分配散列表以便选择同一服务器为同一键值提供服务？原始方案可能是这样的：

$$s = H(k) \bmod n$$

式中，$H(k)$ 是散列函数；k 是键值；n 是服务器的数量；s 是服务器标签，它等于 $H(k)$ 除以 n 的余数。

只要 n 是常数，该方案就可以正常工作，但当服务器数量动态增加或减少时，它很可能会出现不同服务器提供服务的情况，这与云计算中常见的情形相同。因此，缓存空间不足，应用性能下降，且必须对最新集群中的所有服务器进行更新。

显而易见，这是不可取的，因而可以采用另一种方案。为此，Memcached 实现方案通常采用一致性散列算法的变体[35] 来实现服务器池变化时所需的更新最小化，并实现键值给定时由同一服务器提供服务的概率最大化。一致性散列计算的基本算法[36] 可以描述如下：

1）将散列函数的取值范围映射到圆上，其中最大值以顺时针方式环绕最小值；

2）为池中的每台服务器赋值（即圆上的点）作为其标识符㊁；

3）为了缓存键值 k 的数据项，选择标识符大于或等于 $H(k)$ 的服务器。

在本章参考文献［36］中，为键值 k 选择的服务器称为 k 的继任者，负责 k 和前一台服务器标识符之间的弧。例如，图 6.39 给出了包含 3 台服务器的圆，其中服务器 1 负责将键值散列计算的关联数据项缓存到 6、7、0 和 1；服务器 3 负责将键值散列计算的关联数据项缓存到 2 和 3；服务器 5 负责将键值散列计算的关联数据项缓存到 4 和 5。

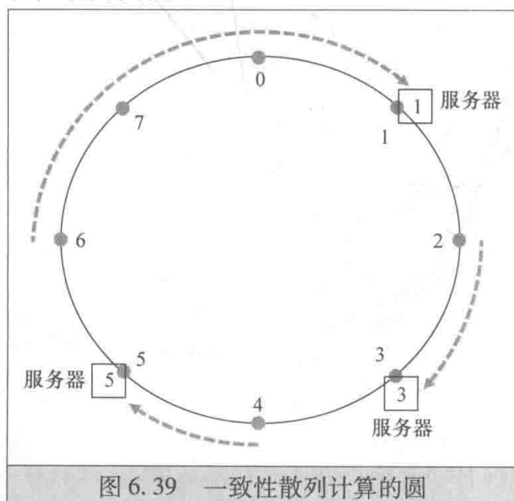


图 6.39　一致性散列计算的圆

一致性散列算法的直接结果是，服务器的离开或加入只会影响到其直接邻居。换句话说，当一台新服务器 p 加入池时，先前分配给原始 p 继任者的某些键值现在将被重新分配给服务器 p，而其他服务器则不受影响。类似地，当旧服务器 p 离开池时，先前分配给它的键值现在将被重新分配给 p 的继任者，而其他服务器则不受影响。在图 6.39 所示的实例中，添加新服务器 7 会导致将键值 6 和 7 重新分配给新服务器，而删除服务器 3 会导致将键值 2 和 3 重新分配给服务器 5。

㊀ 严格来说，最近不常用的数据项就是这里所要表达的意思；最近最少使用的数据项只有一项。

㊁ 可以通过散列函数完成赋值，该函数可能与 H 不同，但取值范围相同。

基本算法允许服务器池进行有效扩展，并为性能进一步提升奠定了坚实的基础。本章参考文献[37]描述了一种用于在服务器之间实现更好负载分配的增强方案。

总而言之，Memcached 被证明是一种能够提升应用性能的有效的可扩展机制。它已经广泛应用于 Facebook、Twitter 和 YouTube 等高流量网站。特别是，Facebook 部署了数千台 Memcached 服务器来支持其社交网络服务，并创建了世界上最大的键值商店，每秒处理超过 10 亿条请求，并存储了数万个数据项[38]。

参考文献

[1] Glanz, J. (2012) The Cloud factories: Power, pollution and the Internet. The New York Times, September 22. www.nytimes.com/2012/09/23/technology/data-centers-waste-vast-amounts-of-energy-belying-industry-image.html? pagewanted=all.

[2] Pianese, F., Bosch, P., Duminuco, A., et al. (2010) Toward a Cloud operating system. Network Operations and Management Symposium Workshops (NOMS Wksps). IEEE/IFIP, pp. 335-342.

[3] SNIA Technical Council (2003) The SNIA shared storage mode. www.snia.org/sites/default/files/SNIA-SSM-text-2003-04-13.pdf.

[4] Ousterhout, J. (n.d.) RAMCloud. Stanford University. https://ramCloud.stanford.edu/wiki/display/ramCloud/RAMCloud.

[5] Dormando (n.d.) What is memcached? http://memcached.org/.

[6] Kant, K. (2009) Data center evolution—a tutorial on state of the art, issues, and challenges. Computer Networks, 53 (17), 2939-2965.

[7] Electronic Industries Alliance (1992) EIA-310-D: Cabinets, Racks, Panels, and Associated Equipment. Electronic Industries Alliance, Arlington.

[8] ISO/IEC (1994) ISO/IEC 7498-1: Information Technology—Open Systems Interconnection—Basis Reference Model: The Basic Model. International Organization for Standardization, Geneva.

[9] Widmer, A. X. and Franaszek, P. A. (1983) A DC-balanced, partitioned-block, 8B/10B transmission code. IBM Journal of Research and Development, 27 (5), 440-451.

[10] Jacob, B., Ng, S. W., and Wang, D. T. (2008) Memory Systems: Cache, DRAM, Disk. Elsevier Science, Amsterdam.

[11] Paulsen, K. (2011) Moving Media Storage Technologies: Applications &Workflows for Video and Media Server Platforms. Elsevier Science, Amsterdam.

[12] Sandberg, R., Goldberg, D., Kleiman, S., et al. (1985) Design and implementation of the Sun network filesystem. Proceedings of the Summer USENIX Conference. USENIX, the Advanced Computing Systems Association, Berkeley.

[13] Nobel Prize organization (2007) Class for Physics of the Royal Swedish Academy of Sciences. The Nobel Prize in Physics 2007. www.nobelprize.org/nobel_prizes/physics/laureates/2007/advanced-physicsprize2007.pdf.

[14] Ghemawat, S., Gobioff, H., and Leung, S.-T. (2003) The Google File System. SOSP '03, Bolton Landing, NY, pp. 29-43.

[15] Doeppner, T. W. (2011) Operating Systems In Depth: Design and Programming. John Wiley & Sons, Inc, Hoboken.

[16] Kleiman, S. R. (1986) Vnodes: An architecture for multiple file system types in Sun Unix. Proceedings of the Summer USENIX Conference.

[17] Birrell, A. D. and Nelson, B. J. (1984) Implementing remote procedure calls. ACM Transactions on Computer Systems, 2 (1), 39-59.

[18] Thurlow, R. (2009) RFC 5531, RPC: Remote Procedure Call Protocol Specification Version 2. Vol. RFC 5531. http://tools.ietf.org/html/rfc5531.

[19] Eisler, E. (2006) RFC 4506, XDR: External Data Representation Standard. http://tools.ietf.org/html/rfc4506.

[20] Dunning, D., Regnier, G., McAlpine, G., et al. (1998) The virtual interface architecture. IEEE Micro, 2 (18), 66-76.

[21] Van Meter, R., Finn, G. G., and Hotz, S. (1998) VISA: Netstation's virtual Internet SCSI adapter. ACM SIGOPS Operating Systems Review (ACM), 32 (5), 71-80.

[22] Meth, K. Z. and Satran, J. (2003) Design of the iSCSI Protocol. Proceedings of the 20th IEEE/11th NASA Goddard

Conference on Mass Storage Systems and Technologies (MSS' 03), IEEE, pp. 116 – 122.

[23] Azagury, A., Dreizin, V., Factor, M., et al. (2003) Towards an object store. Proceedings of the 20th IEEE/11th NASA Goddard Conference on Mass Storage Systems and Technologies (MSS' 03), IEEE, San Francisco, CA, pp. 165 – 176.

[24] Azagury, A., Canetti, R., Factor, M., et al. (2002) A two layered approach for securing an object store network. Proceedings of the First International IEEE Security in StorageWorkshop, IEEE, San Francisco, CA, pp. 10 – 23.

[25] Factor, M., Nagle, D., Naor, D., et al. (2005) The OSD security protocol. Proceedings of the Third International IEEE Security in Storage Workshop, IEEE, San Francisco, CA, pp. 11 – 23.

[26] Troppens, U., Erkens, R., Mueller – Friedt, W., et al. (2011) Storage Networks Explained: Basics and Application of Fibre Channel SAN, NAS, iSCSI, Infiniband and FCoE. John Wiley & Sons Ltd, Chichester.

[27] Vaghani, S. B. (2010) Virtual machine file system. ACM SIGOPS Operating Systems Review, 44 (4), 57 – 70.

[28] Smoot, S. R. and Tan, N. K. (2012) Private Cloud Computing: Consolidation, Virtualization, and Service – Oriented Infrastructure. Morgan Kaufmann, Waltham, MA.

[29] Harari, E. (2012) Flash memory—the great disruptor! In Winner, L. (ed.), IEEE International Solid – State Circuits Conference Digest of Technical Papers (ISSCC). IEEE, San Francisco, CA.

[30] Hennessy, J. L. and Patterson, D. L. (2012) Computer Architecture: A Quantitative Approach. Morgan Kaufmann, Waltham, MA.

[31] Computer History Museum (2012) Oral history of Fujio Masuoka, September 21. http://archive.computerhistory.org/resources/access/text/2013/01/102746492 – 05 – 01 – acc.pdf.

[32] Hu, X. – Y., Eleftheriou, E., Haas, R., et al. (2009) Write amplification analysis in flash – based solid state drives. Proceedings of SYSTOR 2009: The Israeli Experimental Systems Conference, The Association for Computing Machinery, New York.

[33] Ousterhout, J., Agrawal, P., Erickson, D., et al. (2011) The case for RAMCloud. Communications of the ACM, 54 (7), pp. 121 – 130.

[34] Fitzpatrick, B. (2004) Distributed caching with memcached. Linux Journal, 124, 5.

[35] Karger, D., Lehman, E., Leighton, T., et al. (1997) Consistent hashing and random trees: Distributed caching protocols for relieving hot spots on the World Wide Web. Proceedings of the 29th Annual ACM Symposium on Theory of Computing, ACM, New York.

[36] Stoica, I., Morris, R., Karger, D., et al. (2001) Chord: A scalable peer – to – peer lookup service for internet applications. Proceedings of the 2001 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications, ACM, New York, 31 (4), pp. 149 – 160.

[37] DeCandia, G., Hastorun, D., Jampani, M., et al. (2007) Dynamo: Amazon's highly available key – value store. Proceedings of the 21st ACM SIGOPS Symposium on Operating Systems Principles, ACM, New York, 41 (6), pp. 205 – 220.

[38] Nishtala, R., Fugal, H., Grimm, S., et al. (2013) Scaling memcache at Facebook. Proceedings of the 10th USENIX Conference on Networked Systems Design and Implementation, USENIX Association, Berkeley, CA.

第7章 »

云内部的运营、管理与业务流程编排

本章标题中的第一个单词是指支持云基础设施的运营、管理和维护（OA&M）的手段。虽然运营和管理的做法已经非常容易理解，甚至部分实现了标准化，但“编排”一词仍然有些含糊不清，这是行业中滥用最多的术语之一。然而，业务流程的概念对云计算至关重要。我们的首要任务就是澄清这一概念。

在19世纪和20世纪，事情更为简单一些㊀，当时编排简单的是指由作曲家执行的任务，为乐器合奏（通常是交响乐团）写乐谱。该词还可以指代为管弦乐队写谱的音乐学科（音乐学校教授的作曲课程的一部分）。该学科记录了各种乐器（弦乐器、木管乐器、铜管乐器和打击乐器）的各种组合代表性音乐特征（音域、音色、技术难度和音调），人们也将该学科称为乐曲研究，它教给人们如何将不同乐器进行组合或并置，以实现由作曲家设想的音色和平衡。需要注意的是，现代交响乐团所使用乐器的物理特性在很大程度上已经实现了标准化，且乐团演奏者已经按照该标准进行了培训。这使得乐曲演奏相当精确（例如，标准规定了哪个颤音易于在长号上演奏，哪个颤音无法在长号上演奏）。另外，编排本身（即将各种乐器的音质加以组合以达到新效果的部分）除了列出一些普适原则以外，还可以从大师的作品中引入各种实例以说明所产生的音效。如果人们以这些实例作为规则，则他们无法产生新的音效。然而，伟大的作曲家（特别是19世纪的Richard Wagner以及20世纪的Maurice Rarel和Igor Stravinsky）通过发现适合他们各自艺术构想的新颖而醒目的声音组合，在编排领域引发一场革命。一旦他们的音乐广为人知，并被人们所接受，则他们的成果成为教授编排的新资料。我们为感兴趣的读者推荐一本与编排有关的优秀书籍[1]，至少有一位作者从阅读中受益匪浅。

云计算中编排的含义与音乐中编排的含义类似。在云中，“乐器”是前几章所描述的资源。“乐器”一词既是指物理资源（即主机、存储设备和联网设备），又是指软件资源（虚拟机管理程序和各种操作系统），所有这些资源都以引入和支持云服务为唯一目的进行“利用”。NIST（National Institute of Standards and Technology，美国国家标准与技术研究院）的两本云架构出版物[2,3]涉及这一主题，并给出如下定义：

“业务流程是指部署系统组件以支持云提供商在安排、协调和管理计算资源方面的活动，这些活动旨在向云消费者提供云服务。”

NIST云计算参考架构[3]描述了业务流程任务，如图7.1所示。

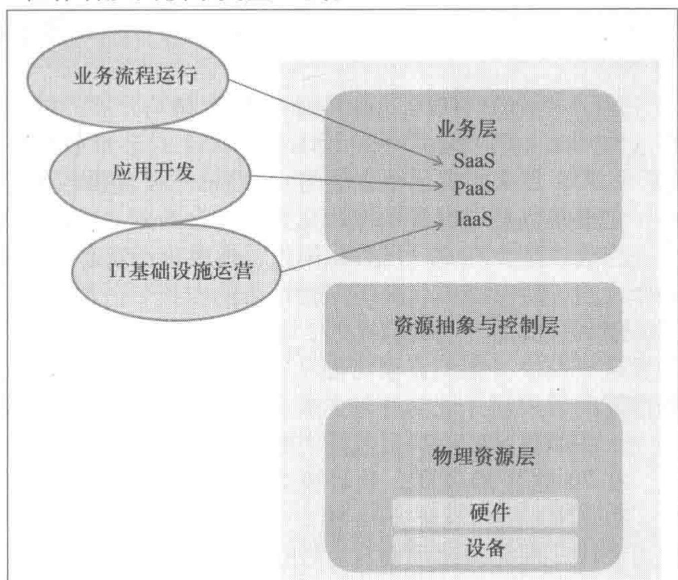


图7.1 业务流程（参照NIST SP 500－292文档）

NIST的3层模型代表了云提供商涉及的组件分组。顶部是用于定义服务访问接口的业务层。中间层是通过软件对物理计算资源进

㊀ 韦氏在线词典将这一术语首次使用的时间追溯到1859年。

行抽象和访问控制的资源抽象与控制层。这里，控制问题与资源分配、访问控制和监控有关。NIST 将其称为“将多种底层物理资源及其软件抽象联系在一起的软件结构，以支持资源池化、动态分配和测量服务”。堆栈底部是物理资源层，它包括所有硬件资源——计算机、存储组件和网络，还有“设备资源，诸如供暖、通风和空调（Heating，Ventilation and Air Conditioning，HVAC）、电力、通信以及其他物理设备”。

本章参考文献［3］中没有规定如何整合这些资源，稍后将在本章提供一些“引擎盖式”的观点（特别是将在 7.3 节中对上述模型进行扩展）。重要的是，NIST 描述强调了云提供商所执行的业务流程和服务管理任务之间的区别。服务管理包括“管理和运行云消费者所要求或提出服务必需的所有与服务相关的功能”。这 3 种服务管理类型包括业务支持、服务提供和配置以及可移植性和互操作性。

在业务领域中，支持类型是客户、合同和库存管理、审计和计费、报告和审计、定价和评级等任务。云的实际操作是服务提供和配置类型的主题，其任务包括配置、资源变更、监控和报告、计量和 SLA（Service Level Agreement，服务等级协议）管理。最后，数据传输、虚拟机图像迁移以及全方位应用与服务迁移可移植性和互操作性的主要任务，可以通过统一管理界面来完成。

为了理解本章实际涉及的内容，有必要对各部分内容进行区分，并分别观察每部分内容的演变。

在本章的 7.1 节中，将讨论企业（即 IT）界业务流程概念的演进，其中业务流程概念实际上是在世纪之交诞生的。

在 7.2 节中，回顾了网络和运营管理学科的发展情况，重点介绍了运营支持系统的演进。需要注意的是，除了“网络”一词外，网络管理是一种纯软件问题。网络管理既适用于企业界，又适用于电信行业，但它最初源于电信世界。在这一讨论背景下，还将介绍几个广泛实施的标准。

7.3 节综合了云情境中的上述概念。在这一情境中，云提供商（自然需要自己的工具来编排这些服务的业务流程）向企业提供托管服务（以及适当的业务流程工具）。可以预期，这里并未太多涉及历史，更不用说标准。但是在撰写本书时，历史正在创造中，且通过多个开放源代码举措加快了历史进程！

7.4 节（最后 1 节）涉及身份和访问管理主题。先前讨论过，现在再次重复：特别是许多人将整体安全和身份管理的成功看作是最重要的事情。当然，十多年来一直从事这一课题研究的作者们非常赞成这一观点。

7.1 企业内部的业务流程

这一术语的起源可以追溯到 20 世纪初期开展的信息技术运动。我们将这一运动称为面向服务的架构（Service - Oriented Architecture，SOA）。正如稍后在本节将看到的那样，SOA 在 2009 年“死亡”（至少 SOA 是从此时开始衰落的），但总体理念和运动目标仍然有效而且还在执行！

主要动机是通过利用模块化并支持分布式处理来打破开发和维护整体式应用[⊖]的旧模式。

当然，至少在 20 世纪 60 年代，模块化一直是软件的圣杯，且在过去几年中已经完成了许多工作。迄今为止，所有结构化高级编程语言的祖先——ALGOL - 60 为模块提供了拥有独立变量命名功能的机制，使其只能通过明确定义的、基于参数的接口实现相互交互。一旦定义了接口，则程序员可以独立开发这些模块（他们甚至可能从不需要彼此交谈）。然后，可以对这些模块进行编译，将生成的对象代码存储在库中，并最终将与主线应用代码链接。这里需要强调的是，只要两个模块都遵循相同的接口，则一个功能更好的模块可能随时替换另一个模块。

在 20 世纪 80 年代，这种模式的演变分为 3 个独立的发展方向，从而影响了业务流程的理念。

由 UNIX 操作系统 shell 接口孕育的第 1 个发展方向，为程序员提供了用于执行一套独立程序的强有力手段，而无须编译或与主线代码建立链接，甚至可以将这些程序进行整理，使某个程序的输出可以载入到另一个程序（通过管道接口）。需要强调的是，与先前其他操作系统提供的作业控制语言（Job Control Language，JCL）环境不同，shell 环境实际上是一个成熟的协作编程平台。任何人都可以编写一条新“命令”并对其进行编译，然后将它提供给其他人。此外，只要将不同模块存储在不同目录

⊖ 本节讨论的任何内容都与应用开发和最终编程有关。

中，则同一“命令”名可以在这些模块之间进行共享。用于获取模块的目录集可由一个环境变量来表示，该变量可以即时进行更改㊀。另外，shell 程序在变更时不需要进行重新编译，因为它们压根儿不需要进行编译——需要对其进行解释。最后，shell 脚本中调用的模块可以用任何语言（包括 shell 本身）编写。

第 2 个发展方向称为面向对象编程，它大大简化了模块（以前被认为是过程）的接口。而先前程序员需要了解过程库运行所需的数据结构的每个细节，采用面向对象编程技术，将数据结构与执行操作的方法（即过程）一起进行了封装。方法只对程序员可见，因而程序员不再需要关心数据结构。数据结构可能会相当复杂㊁，但使用对象的程序员不需要理解这种复杂性；只有那些能够实现对象类的程序员（相当于用于定义数据结构的类型）才需要这么做。实际上，可以将针对给定类的实例化对象看作是服务㊂。第一种面向对象的语言——SIMULA，实际上是在 1967 年开发的（与 ALGOL - 67 的开发时间大致相同），且 SIMULA 是 ALGOL 的一个自然超集，仅用于模拟系统的目的（实际上，它已用于模拟复杂的硬件系统）。由于每个系统都是由“黑盒子”组成的——同一种类型（或按照面向对象的说法：类），因而这种范式是与生俱来的。当然，SIMULA 的目标是建模，而不是有效的代码重用。将 SIMULA 标准化的工作花了约 20 年，该项任务由 SIMULA 标准化小组 1 负责实施，并于 1986 年完成。当 SIMULA 标准化工作结束时，1983 年，贝尔实验室研究人员 Bjarne Stroustrup 博士发布了一种新语言——C + +，Bjarne 自 1979 年以来一直在默默工作。C + + 从 SIMULA 借鉴了很多东西，但它是基于（且实际上是编译到）C 语言的，而 C 语言是设计作为系统编程语言的（或者换句话说，它支持程序员尽量走捷径，以便与现有硬件紧密合作）。正是这种效率（结合面向对象范例的完全实现）使得 C + + 倍受欢迎。国际标准化组织（International Organization for Standardization，ISO）已经批准将 ISO/IEC 14882 确定为 C + + 标准。1998 年，国际标准化组织发布了一个早期版本，现行标准是 ISO/IEC 14882：2011，业界将其称为 C + +11。关于这个问题，强烈推荐一本由 C + + 发明人和首位开发人员出版的好书[4]。C + + 也是其他众多流行的解释型面向对象语言（尤其是 Java）的先驱。Java 主要设计用于迅猛发展的轻量级应用开发（在系统编程方面，C + + 仍然占据统治地位）。

从程序员的角度来看，新型面向对象的语言已经实现了参数多态性［一种支持程序员使用灵活数量参数和（在某种程度上）灵活类型参数来定义子程序的特征］。这大大提高了使用服务的程序与提供服务的程序之间的接口灵活性。我们将此类接口称为应用编程接口（Application Programming Interface，API）。

第 3 个发展方向是分布式计算。强烈推荐本章参考文献［5］这本内容丰富的综合类专著。作为发展方向的一部分，人们围绕远程执行方式开展了大量研究和标准化工作。这里，主要目的是使程序员避开跟踪计算资源实际物理分布的复杂（且一般非常乏味）细节的困扰。为此，程序员甚至不需要知道计算资源的实际物理分布情况。为实用起见，API 必须与操作系统或任何应用库已提供的 API 完全一致，此类接口属于过程调用接口（或在面向对象的模型中，此类接口属于方法调用接口）。

鉴于此目标，人们开发出了远程过程调用（Remote Procedure Call，RPC）模型。在这一模型中，程序员编写一个本地过程调用（除了本地程序调用之外，还有其他什么调用呢?）但是，底层软件通过应用协议将调用“传输”给另一台机器。该模型主要适用于客户端/服务器的交互。在此过程中，客户端程序调用服务器上的“远程”过程。这里的问题是无足轻重的（考虑通过地址将参数从客户端传输到服务器，或者崩溃恢复，特别是在服务器端）。

除了涵盖并发执行的算法部分之外，人们已经开发出由跨机器对象提供的、用于广播服务㊃以及用于访问此类服务的工业基础设施。为此，人们已经开发出多种基础设施，因为涉及诸多标准化组织和

㊀ 遗憾的是，协作混搭的易实现性与安全性是背道而驰的。便于实现协作的非常巧妙的机制引入了重大安全问题。某个人编写一个致命程序（许多人实际上已经这么做了），赋予它一个常用名称，并将其放在由环境变量 PATH 指定的目录中。除非每个模块重置此变量（或在进程开始之前对模块仔细检查），否则不敢保证它调用的模块是它们应调用的。同样的问题仍然存在于支持同构模块的框架中。

㊁ 考虑一个复数相乘的过程，该过程支持每个参数要么表示为（Re，Im），要么（以三角形式）表示为（模，辐角），这样当程序员需要时，他们可以混搭使用表达式。

㊂ 根据前面的脚注，复数运算（加法、乘法、除法等）就是此类服务的一种实例。

㊃ 又一个被滥用的术语。从严格意义上讲，它表示给定服务器上所支持的方法（即对象上的操作）。

论坛。这里仅举几例，这些标准化组织和论坛包括 ISO/IEC、ITU - T（International Telecommunications Union Telecommunication Standardization Sector，国际电信联盟电信标准分局）、对象管理组（Object Management Group，OMG）以及（后来伴随万维网成功而成立的）万维网联盟（World Wide Web Consortium，W3C）和结构化信息标准促进组织（Organization for the Advancement of Structured Information Standards，OASIS）㊀。

20 世纪 90 年代中期形成的通用模型如图 7.2 所示。

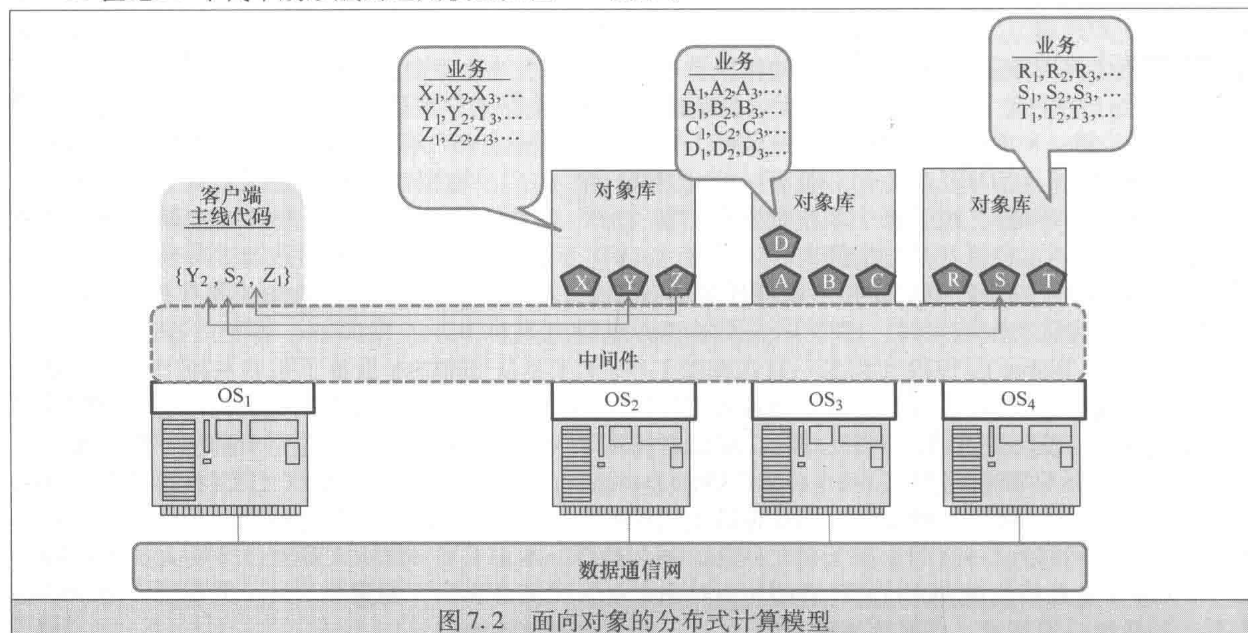


图 7.2　面向对象的分布式计算模型

这里，客户端程序可以远程调用跨基础设施的各种对象方法。基础设施可能包括运行不同操作系统的不同机器。“物理”性质的唯一要求是这些计算机能够通过数据网络进行互连。

当然，每个对象库都可以用自己的语言来实现。对于原始语言和要执行相应代码的操作系统来说，远程调用机制大体上是不变的。这可以通过中间件来实现，因为中间件能够提供自己的原语（本地 API）来将编程环境与操作系统隔离开来，从而确保通用可移植性的实现。由于环境还支持对象库发布其服务，因而一些模型包含了服务代理的概念。服务代理的工作是负责匹配客户端服务需求与各种服务提供商的库。

20 世纪 90 年代中期，当人们都在关注互联网和万维网时，出现了一种令人不安的发展势头，即大家开始拒绝接受先前已经完成的标准（以及先前已经完成的支持基础设施的大多数标准）。这对于从事标准化研究的人来说并不一定是坏事，他们突然得到了令人兴奋的新工作；对于那些如雨后春笋般迅速成长起来并替换掉旧标准化组织的标准化论坛来说也没有坏处。这些旧标准化组织始终基于几乎一夜之间被打上“旧”标签的技术努力完成标准的制定。对于那些在如何“简化”编程方面有着新观念的人也是有好处的，因为时尚潮流推动许多未经检验的想法进入标准。技术在快速发展，诸多公司（以及依赖于这些公司的论坛）面临着风险。最终，当泡沫破裂时，它们被历史所遗弃，但即使对于技术本身，未经检验技术的快速发展导致的后果也是毁灭性的！

问题的根源是对分布式处理技术新浪潮的真正膜拜：人们相信所有应用层协议必须是基于 ASCII（American Standard Code for Information Interchange，美国信息交换标准代码）文本的。事实上，简单邮件传输协议（Simple Mail Transfer Protocol，SMTP）的确是基于 ASCII 文本的，这在大多数终端属于电传类型的时代是非常先进的，且使用 ASCII 文本有助于测试和调试（当然也有利于黑客入侵，虽然这绝对不是设计目标！）。类似地，Web 的主要协议——超文本传输协议（HyperText Transfer Protocol，HTTP），最初只需处理采用 ASCII 编码（HTML）的文件传输问题。由于协议相关数据量比有效载荷小，

㊀ 在电信行业，电信信息网络架构协会（Telecommunications Information Networking Architecture Consortium，TINA - C）在 1993 ~2000 年间开展了一系列标准化和研发工作，贝尔实验室的作者们为此做出了重大贡献。本章参考文献［4］提到，TINA - C 的结果在业界引发了人们的极大兴趣。当前，有一本专门介绍 TINA - C 的图书[6]。

因而基于文本的编码非常合理。但这些看似必要的决策（或者至少在 SMTP 和 HTTP 开发时都是正当的）后来莫明其妙地被解读为所有互联网应用协议都必须采用文本编码的准则。该准则很快成为一种信仰，并加入了其他虚假信仰（如 IPv6 比 IPv4 更“安全”）。随着应用协议的增长，使用该准则变得越来越荒谬。不仅数据量巨大，而且对实时协议来说，解析数据变成了一个大问题。事实上，IETF（Internet Engineering Task Force，互联网工程任务组）的 HTTPbis 工作组当前正在开发的新版 HTTP[7] 使用了二进制编码，并针对这一变化提供如下说明：“... ... HTTP/1.1 报头域通常是重复和冗长的，除了生成更多或更大的网络数据包之外，还可能会导致小型初始 TCP 拥塞窗口快速填充... ... 最后，这种封装还通过使用二进制消息分帧来支持消息的可扩展处理”。

文本编码对面向对象的分布式计算的影响首先体现在放弃了抽象语法标记（ASN.1）编码标准（它需要编译成二进制格式㊀），转而支持可扩展标记语言（Extensible Markup Language，XML）㊁。XML 本身不存在任何错误，但滥用它可能会产生灾难性后果㊂。尽管 HTTP 本身提供了远程 API 访问机制，但 W3C 决定开发一种运行在 HTTP 之上的 RPC 机制。因此，一种新协议——SOAP（Simple Object Access Protocol，简单对象访问协议）应运而生。事实证明，“简单”这一单词并不准确，因而在 SOAP 1.2 版本中不再使用首字母缩略词的展开版。SOAP 变得相当时尚，且人们在 SOAP 之上已经开发出了复杂的 SOA 基础设施。

虽然 SOAP 过去（且现在仍然）被用作远程过程调用机制，但 XML 格式的序列化使其执行效果要比 OMG 开发的通用对象请求代理体系结构（Common Object Request Broker Architecture，CORBA）中的 RPC 要差得多。SOAP 嵌入二进制对象时单独需要额外工作（和额外标准），但已证明更为糟糕的事实是，由于 SOAP 采用 HTTP 作为传输协议，因而它面临着与 HTTP 直接竞争的问题。严格来说，虽然 SOAP 运行在 HTTP 之上不是必需的，但是默认 SOAP/HTTP 绑定成功，部分原因是这样可以确保穿越防火墙（读者可能还记得前面提到的 4 月 1 日发布的 RFC，这是一个相当愤世嫉俗的笑话实例，却在转眼间成为现实）。结果不仅仅是政治对抗（没有人希望他或她的应用协议仅是他人应用协议的一种传输工具!），而且也是一种困境：要么接受 HTTP 严格的客户端/服务器结构（在这种架构中，每次通信均必须由客户端启动，从而使得服务器通知无法实现㊃），要么发明越来越多的机制来弥补这一限制条件。但是，一般来说，不利于 RPC 方法（特别是 SOAP）的最重要论据就是作为一种概念的远程过程调用无法轻易适应 Web 结构，这涉及 midboxes（代理和缓存）。事实上，业界发明了更多的机制，增加了更多的复杂性。

最后，业界曾使用被称为表示状态转移（Representation State Transfer，REST）的“本地”Web 学科组织过一次反对 SOAP 的活动。REST API 至少可获得 Web 访问权限。本书将在附录 A 中详细讨论 REST 原理。目前，考虑到 REST API 中的 API（这是一个有些使用不够妥当的名称，因为它不涉及过程调用本身），我们只提及转向 REST 风格变得非常必要。相反，程序员编写应用层协议数据单元（PDU），为了实现所有实际用途，采取的协议应当是 HTTP。正如稍后会看到的，不存在 REST 标准，REST 只是一种风格。

谈到标准，CORBA 已经拥有，SOAP 和其他几项技术也有标准，但头条新闻较少。人们不应忘记安德鲁·塔南鲍姆（Andrew Tanenbaum）的格言：“标准的好处是让你拥有更多的选择!”

在 REST 与 RPC 的情况下，由于在企业中出现的 3 层架构得到广泛实施，并成为提供软件即服务（Software as a Service，SaaS）的首选模式，因而劳动分工非常简单。在这一模型中，第 1 级（客户端）

㊀ ASN.1 已在 ITU-T 建议书的 X.680 ~ X.695 系列中进行了定义，ASN.1 是 20 世纪 90 年代初开发的，旨在克服协议定义中指定位串的问题。使用 ASN.1，编码方式在 PASCAL 类数据结构中指定，然后根据二进制编码规则进行编译。实际上，编译是一个额外步骤，但总体方案提供了一种高效二进制编码方式。

㊁ W3C 已开发出了 XML。它是一种基于文本的通用编码方案。在使用不合理的情况下，会生成“冗长”的输出，但没有人定义怎样才算是合理使用 XML。

㊂ 这里将给出的是作者的亲身经历，自 2003 年以来，马丁·汤姆森一直使用一种流行音乐编写程序。这种将乐谱编辑与即时播放结合起来的工具非常有用，但自从使用 XML 来编码声音的新版本开发出来后，该工具运行速度非常慢（有时达到无法响应的地步），从而导致系统经常崩溃。最终，作者不得不购买一台更强大、更昂贵的计算机，来确保能够使用以前版本中完美运行的相同功能。

㊃ 想象一下使用轮询机制而不是中断机制。

负责发布基于HTTP的查询，第2级（服务器）负责提供业务流程逻辑和数据访问，而第3级（通常是一组运行数据库软件的主机）负责提供实际数据。客户端使用的是REST范例来访问服务交付基础设施的前端（第2级），而后端通信则可以使用RPC和其他分布式处理机制。

在图7.3所示的实例中，客户端请求来自于第2层服务器的特定雇员的工资单存根。反过来，服务器在查询过包含工资单记录和考勤报告的公司数据库后生成表单。当然，该行为只有在对请求了该信息的用户进行认证并确保授权用户接收后才能执行。另一个例子是当前无处不在的基于Web的电子邮件服务。Web客户端与第2级服务器采用的是REST风格，而第2级服务器使用实际的邮件客户端协议向SMTP邮件服务器发送电子邮件，并接收来自SMTP邮件服务器的电子邮件。

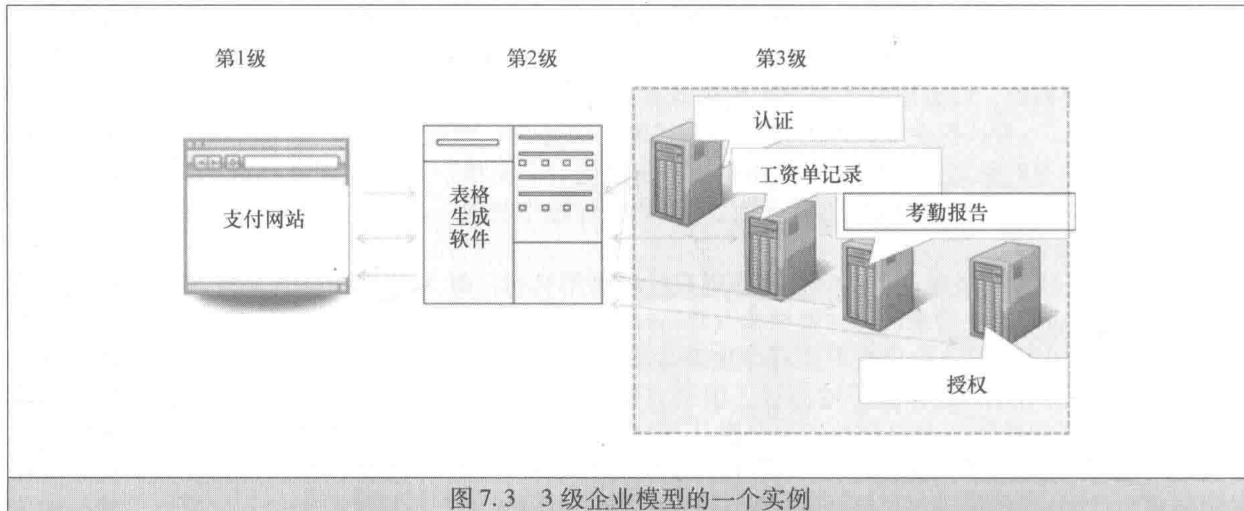


图7.3　3级企业模型的一个实例

迄今为止，除了竞争标准和非互通实现方案之外，架构和机制所带来的模块化优势是显而易见的：没有什么能比提供一批即用型执行业务模块的基础设施更加模块化，这些业务模块超越一切，可以从任何地方调用。但它的承诺更进一步，有望通过降低编程难度来降低IT成本，以支持定义业务的那些人来编程实现，从而消除他们对机构内部专业软件开发的依赖。

兑现承诺的业界努力正在以SOA的形式呈现，而SOA中引入了业务流程这一术语。

7.1.1 面向服务的架构 ★★★

首先注意到，业界对SOA的含义存在诸多误解。托马斯·埃尔（Thomas Erl）在其权威专著[8]的开头写道："在我印象中，没有任何其他术语比'面向服务'更易引起混淆。其明显的歧义已经导致供应商、IT专业人员和媒体宣称自己的解释是正确的。当然，这使得准确把握标签为'面向服务'技术架构的含义更加困难。"

这正是问题所在：如何解释SOA。当然，人们老生常谈的是，愿景（在SOA的情形中，愿景是在分布式计算平台上远程执行API定义的服务）能够以不同方式实现，这不一定需要彼此之间相互作用。然而，一旦发现人们对某件事情进行了详细规定以确保解释唯一，则它有可能冒着被贴上"实现方案"标签的危险。

为此，SOA规范正在持续制定。首先，W3C生成了用于将网络服务描述为一组端点的基于XML的Web服务描述语言（Web Services Description Language，WSDL），这些端点运行于包含面向文档或面向过程的消息之上。"WSDL应当是抽象的、可扩展的，这样它可以绑定到任何协议上，但该规范重点关注一次绑定上——尤其是在HTTP的子集上与SOAP 1.1进行绑定。

下一个必要的SOA组件（支持发布和后续发现的注册表）标准是由OASIS以统一描述、发现和集成（Universal Description，Discovery and Integration，UDDI）标准的形式（也能以基于XML的形式）开发的。

然而，另一组SOA组件涉及服务质量（在这种情境中，这一概念与数据通信中的QoS无关）问题，它还包括一组与安全性［建立在OASIS安全断言标记语言（Security Assertion Markup Language，SAML）标准之上］、可靠性、策略断言和业务流程本身有关的参数。后者的标准——Web服务业务流



程执行语言（Web Services Business Process Execution Language，WSBPEL）是由 OASIS 根据 IBM、微软和 BEA 共同努力制定的早期规范生成的，并受到 IBM Web 服务流程语言（Web Services Flow Language，WSFL）和微软 XLANG 的启发。

简而言之，WSBPEL 使用 XML 编码工具来指定业务流程要求，其方式类似于专门编程语言中指定并发进程执行所采用的方式。两者都提供了用于描述并行活动和处理异常情况的工具。人们针对管理和协调问题还制定了一些其他 Web 服务规范，并将其作为广义服务质量学科的组成部分。

遗憾的是，SOA 的努力结果并未达到预期。到 2005 年底，UDDI 标准只有 400 多页——只有几个开发者有时间来处理这些问题，因为这些规范充斥着不必要的新术语，许多标准化文档也是如此。2005 年 12 月，《SOA 世界》杂志发表了一篇文章，对 IBM、微软和 SAP 关闭其 UDDI 注册管理机构的决定进行了评论。

2009 年 1 月 9 日，伯顿集团分析师安妮·托马斯·马内斯（Anne Thomas Manes）在其博客中写了一则说明，宣布 SOA 已"死"。文章援引了经济衰退的影响（以及 IT 组织拒绝向 SOA 投入更多资金的现实），马内斯女士强调："SOA 疲劳已经转变为 SOA 幻想。商界人士不再认为，SOA 将会带来惊人的收益。"虽然博文并无嘲讽之意——但它将此种情形描述为"IT 行业的悲剧"，因为"面向服务是数据和业务流程快速整合的先决条件"，且表达了为 SOA"幸存者"——Web 混搭和 SaaS 开发面向服务的需求。为此，博文实际上暗示 SOA 这一术语已死，而"面向服务架构的需求要比以往任何时候都要强烈"。

这是当时行业的普遍看法。当某个作者使用 Google 搜索"SOA 为什么失败"时，可以得到超过 300 万个结果。在业务方面，人们将责任归咎于 IT 行业缺乏变革的决心。反过来，商界人士又指责 SOA 的支持者未能充分表达出 SOA 对业务的重要性。"人才短缺"是另一种解释，当然还有更多的解释。

在我们看来，SOA 的历史与 20 世纪 80 年代后期的 OSI 历史类似。事实上，OSI 和 SOA 的命运至少在 3 个方面非常相似，其中之一就是二者都产生了合理的隐喻和基础架构，并在时间的考验下幸存下来。第二个相似之处是 OSI 和 SOA 标准都受到互联网社团的挑战。正当人们宣告基于 SOAP 的 SOA 死亡时，REST 范例正在复苏。第三个相似之处是事情完成方式的根本变化：互联网连接的专用网和互联网支持下的组网部分外包；以及云支持下 IT 服务外包的出现。

4 年后，在《信息世界》杂志的一篇文章中，大卫·林迪卡姆（David Linthicum）指出，SOA 实践在云计算领域中是绝对必要的。也许 21 世纪 SOA 面临的问题是 SOA 具体解决方案的问题。此外，前面提到的 SOA 与企业内的应用开发有关。在云情境下，需要一个更为宽泛的定义。这就是在本书中选择不对 SOA 进行详细描述的原因。工作流就是幸存的 SOA 主要概念之一。

7.1.2 工作流 ★★★

在描述某项任务（任意任务）时，人们可能会列出所有涉及任务完成的活动。其中的某些活动可能并行进行，其他活动需要等待先行活动完成后再开始执行。工作流是一种用于定义和排列任务中所有活动的规范。自然地，为了实现分布式系统中任务的自动化，必须对其工作流进行定义，以确保任务在分布式环境中可执行。

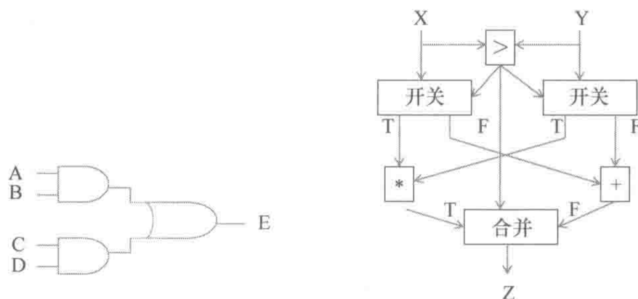
在某种程度上，计算的整个开发是基于工作流的。硬件基于逻辑设计的规则来构建，通过连接执行基本操作的构建块（逻辑门）来处理构成电路。图 7.4a 对这种电路进行了描述。

20 世纪 80 年代，人们掀起了一场研究运动，旨在通过基本块来构建基于工作流的计算机（后来人们称其为数据流机）。这些基本块按照有向图进行链接，且每个块在接收到消息（令牌）时被激活。一份 1986 年的麻省理工学院（Massachusetts Institute of Technology，MIT）备忘录[9]描述了开发这种机器涉及的模型和问题。图 7.4b（参见本章参考文献［9］）给出了数据流机的一个实例，该机器主要用于计算条件表达式。

在某种程度上，数据流机是嵌入到硬件的工作流。如果业界没有意识到实现了标准化的泛在计算平台能够提供更加经济（和更加灵活）的软件实现方法，那么它可能以这种方式进行开发。技巧是开发可以混合和匹配的软件构建块，这与硅片的功能类似。

在引入结构化计算机语言之前，人们使用流程图来确定算法。流程图适用于单一过程规范，但在

描述分布式处理中的并行活动方面显得力不从心。这就是数据流机（也是工作流的所有内容）基于软件的实现方案的用武之地。



a)用于计算$E=(A \wedge B) \vee (C \wedge D)$的逻辑电路　　b)用于计算$Z=\text{if}(X>Y)\text{then}(X*Y)\text{else}(X+Y)$的数据流机

图 7.4　基于流的计算实例

举个例子，这是作者的个人经历，即电话中的服务创建。本书已经提到的智能网络技术是在 20 世纪 80 年代后期到 20 世纪 90 年代开发的。其主要目标是支持电话服务的快速开发。“快速”意味着这些服务开发人员（虽然他们不知道网络结构和处理的分布式性质）可以在图形接口的协助下，通过简单链接图标将服务整合起来。每个图标代表一种独立于服务的构建块（如 queue_call 或 translate_number）。因此，包含基于时间和位置的转换、播放公告和收集输入等复杂的 800 号码服务，可以在数分钟内完成编程。当然，每种独立于服务的构建块的执行本身就是一项复杂的分布式活动。然而，由于它包含在准备就绪的模块中，因而服务程序员并不关心其复杂性。用现代术语来说，可以将每项服务编程为工作流。人们甚至偶尔尝试通过计费和收费过程来协调呼叫建立。由于没有开发标准，因而存在着若干个服务创建环境，但合并服务创建环境并非微不足道。在 20 世纪 90 年代中期，作者研究了 AT&T 公司中统一若干个服务创建环境的方法，本章参考文献［10］对结果进行了报道。

图 7.5 阐明了工作流规范和执行的一般概念。在左侧，工作流程序可表示为活动的有向图。这看起来几乎就像一个流程图，虽然正如书中将要说到的，其实二者存在着显著的区别。具体来说，活动 1 是启动工作流的第一项活动。每项活动（除了最后一项活动和最后一台终端）都拥有指向下一个活动的输出，也就是说消耗该链中的输出。原则上，活动可以形成回环，虽然图中并未给出这样一个实例。

一旦某项活动完成，则可以通过条件测试来选择下一项活动。在实例中，测试块决定活动 2 是否在活动 1 之后开始启动并消耗其输出。如果活动 2 没有启动，则两个活动（活动 3 和活动 4）将同时执行。该实例表明，在将活动 3 和活动 4 对应的输出过滤到活动 5 之前，通过使检查点（CP）等待两项活动的完成，可以实现两项活动的同步执行。支持并发使得工作流规范与流程图规范截然不同。另一个区别是工作流规范维护其显式状态（在左侧的一个块中表示），它可以由所有活动进行读取和更新。

迄今为止，本书仅对工作流规范进行了讨论。执行是一件完全不同的事情，它可由图 7.5 的右侧部分进行说明，该部分对活动 1 进行了扩展。可以看出，它是由 3 个进程——A_{11}、A_{12}和 A_{13}执行的，且这些进程运行于 3 台不同的机器上。然而，状态数据库被保存在另一台机器上（在这一实例中）。当然，在不同主机上运行 A_{11}、A_{12}和 A_{13}不重要——它们可以分布在两台主机之间，甚至可以在同一台主机上运行。它们是进程而不是单个进程中的线程也不重要。这种安排的巧妙之处在于执行主机的选择和执行的形式是绝对灵活的——这些全部留给运行环境来决定。类似地，只要状态数据库满足性能要求，则其位置（实际上也可能是分布式的）是不相关的。为了提高可靠性和系统性能，可以复制状态数据库。很快，我们将会在 OpenStack 设计中应用这一原则。

提高性能的另一方面是工作流优化。如果人们正式定义了工作流规范语言（这样可以对其进行解析），则可以应用编译器理论来消除并行活动的冗余，更重要的是优化并行活动的调度。但是，在工作流重复的情况下，通过统计分析工作流性能是可以实现的，这样能够发现性能方面存在的问题。将图 7.6 描述的这种方法称为路径分析，它在实现诊断工具的工作流中特别有用。



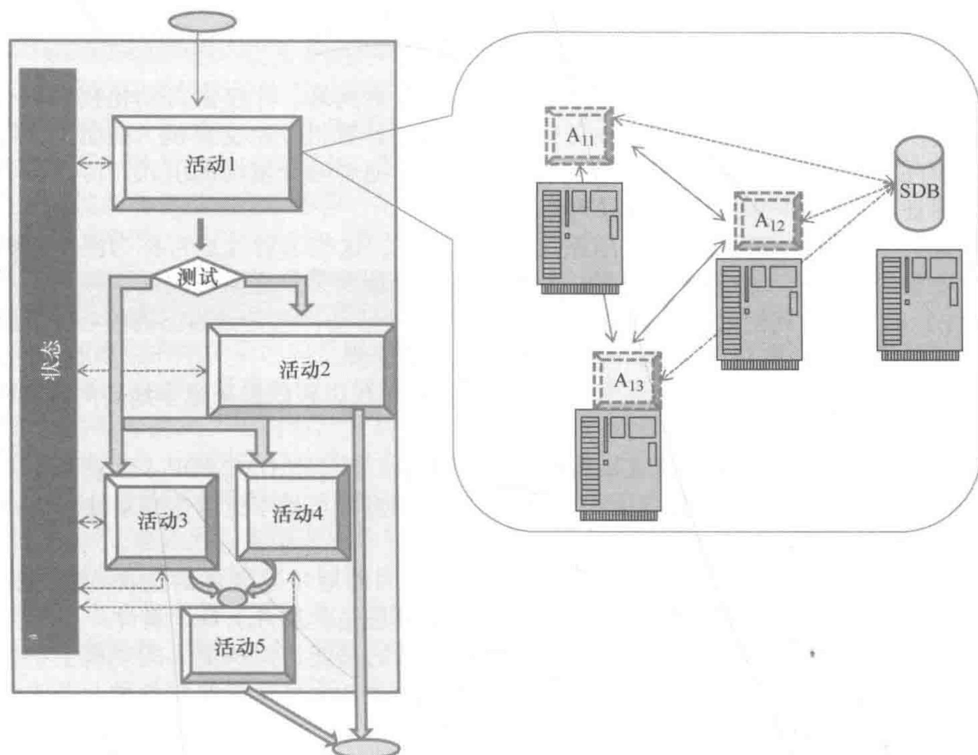


图 7.5　作为活动有向图的工作流

从输入采集的工作流中的某个地方开始，可能存在着实现结果的若干种方式——用通过工作流图的多条路径来表示。通过对路径 A、路径 B，…，路径 Z 之间的推断，发现路径 B 在执行时间方面最优，因而可以将这一条好建议提供给工作流设计人员，并建议他们删除其他路径以简化工作流。

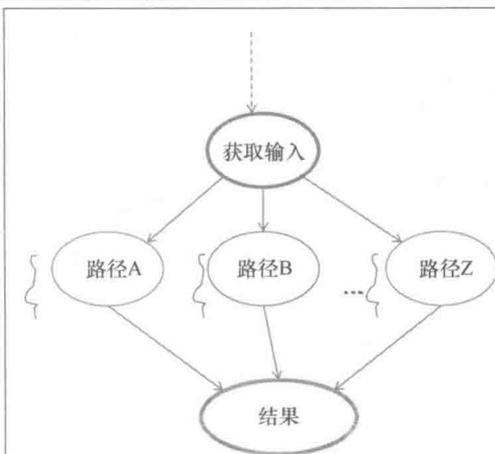


图 7.6　路径分析

目前，存在着大量与工作流相关的文献和产品。这里仅举几例。

本章参考文献［11］对一个与电网相关的早期研究项目——GridAnt 进行了描述，并对当时存在的商用产品进行了综述。本章参考文献［12］对几种工作流优化算法进行了概述，并提出了一种扩展算法（布尔验证算法），用于处理包含条件分支和循环的工作流。

就产品而言，专门网站对 Microsoft Windows Workflow Foundation 进行了描述，该网站还包含诸多质量上好的辅导材料。

亚马逊提供了简单工作流服务（Amazon Simple Workflow Service，AWS）API 和工作流框架，用于从用户程序中调用这些 API。开发人员需要指定协调逻辑（排序、计时和失败响应）以及工作流每一步骤的代码。为此，亚马逊还提供了一种支持协调逻辑的常用编程模式库。

迄今为止，本章已经讨论了应用中工作流的一般用法。在本章的后面各节，将重新回到这一主题，但是将把讨论的焦点缩小到云业务流程中特定任务的应用工作流上。然而，在讨论这些问题之前，需要回顾网络和运营管理的相关概念和技术。

7.2 网络和运营管理

如前所述，网络管理学科早于数据通信学科。它始于电话网络，并仅由自动化技术驱动发展。随着电信网络设备从人工开关面板演进到自动处理呼叫请求的计算机控制交换机，控制个人呼叫的需求正逐步被对控制呼叫的设备进行控制的需求所替代。此外，随着时分复用的引入，传输设备本身的运行复杂到足以保证对活动进行实时监控和管理。

与所有其他大型电话公司类似，在贝尔系统公司[13]中㊀，这些活动（通常称为网络相关业务）是整个公司运营业务的一部分，包括向客户提供服务、业务管理和维护运营。顺便说一下，由于纯历史（甚至是历史性）的原因，我们没有提到这些活动——但这些活动仍然是云当前的核心。有趣的是，今天看到的大量软件技术概念被开发出来，主要用于简化网络运营。

管理进程最初是人工执行的，但是在魔幻的20世纪70年代，它们越来越多地转向计算机处理。人们开发了独立系统——每一系统对应于待管理的一种设备。

最初，贝尔系统公司采购各种主机来承载运营支撑软件，但是当DEC PDP-11微型机出现后（当时最初针对PDP-11开发的UNIX操作系统也已经出现），微型机可用于运营支撑系统（Operations Support System，OSS）㊁的开发。

最终，由于UNIX操作系统可以在任何计算机上运行，因而对特定硬件的选择变得越来越无关紧要。20世纪80年代后期，贝尔实验室㊂软件研发的绝大部分是完全致力于OSS设计。

运营支撑系统或多或少地需要具备与任何企业管理相同的功能，但从更大的规模上讲，因为电信领域包含数千件基于自主计算机的设备（更不用说数亿根独立电话线）以及对各种业务流程和美国政府法规的深度控制，这对运营支撑系统提出了更高的要求。

20世纪80年代，运营支撑系统的发展目标是拥有能够管理所有活动且具有较高量级的普适运营支撑系统㊃。

首先，公司的业务活动是不连续的。当要求提供电话服务时，必须由营业厅进行处理。的确，在一个或另一个数据库中创建了一条客户记录——很可能在几个数据库中创建了客户记录，但是它无法自动到达本地交换机的数据库。

根据贝尔实验室当时的轶事，交换机管理系统的操作人员通过专用终端进行访问，需要转动转椅使用另一终端来登录订单系统，读取客户订单记录，然后转过来将信息重新输入到交换系统中。显而易见，这就是“转椅一体化”这一术语的含义。

准确地说，独立运营支撑系统（在很大程度上仍然存在于电信世界之中）包含如下内容：

1）集成有记录保存系统的主干网。

2）插件库存控制系统。

3）综合布线信息系统㊄。

4）全部网络数据系统。

5）交换控制数据系统。

6）中央办公设备工程系统。

7）诸多设备网络规划系统。

此外，20世纪70年代，AT&T开发了一种中央网络管理系统，该系统在新泽西州贝德明斯特市网络运营中心展示。这里，网络上的所有更新都可以显示在美国墙壁大小的地图上，用于显示网络的状态。必要时，工作于个人终端设备的网络管理员可以采取纠正措施。这是针对网络（与网元）管理做

㊀ 据作者的回忆，20世纪80年代，每名贝尔实验室员工在他/她工作的第一天都会收到一部令人印象深刻的900寻呼机。有趣的是，30多年后，这本书在技术上对历史学家之外的人仍然是相关和有用的。读者仍然可以在书店和网上获得这些信息。

㊁ AT&T自身的3B计算机生产线主要应用于电话交换机领域，尽管它最小的后代（UNIX PC）是普通个人计算机。

㊂ 第59区实验室。

㊃ 今天仍旧如此，但需要明确的是，编排的目标是什么以及如何解决这一问题。

㊄ 与企业客户有关。



出的第一个决定性措施。20 世纪 80 年代后期，AT&T 网络系统部门开发了中央网络流量管理系统，并将其出售给区域和国外运营公司㊀。

回到统一的 OSS 愿景上。这其中的主要障碍是，在贝尔系统中，多种系统独立演进，却没有任何公共平台㊁。重写所有这些软件是不成问题的，但即使做出重写决定，仍然不存在不同厂商可以共同实施的标准。由于该愿景是围绕综合业务数字网（Integrated Services Digital Network，ISDN）技术构建的，该技术综合了电话服务与数据通信服务，第一步是（自然）将数据通信网络的管理整合起来。数据通信网络管理已经发展为自己的一门学科，它以 5 项内容框架作为起点启动 ISO OSI 网络管理项目。这一框架仍然是包罗万象的，本书将在下一节对其进行描述。

7.2.1 OSI 网络管理框架和模型 ★★★

OSI 网络管理框架的第 1 项内容是配置管理，且涉及诸多参数，这些参数值需要在网络中所有设备上维持在指定范围。一些参数值只能由网络所有者直接进行修改，而其他参数是只读的㊂。

OSI 网络管理框架的第 2 项内容涉及故障管理。广义上的“故障”一词是指网络中的任何异常情况。这里，面临的一项大型设计任务当然是对所有事件进行清晰的定义，而这些事件对应于从“异常”到正常的变化。另一项设计任务是一方面选择那些有检测价值的事件，另一方面要确保这些事件的报告不会超出系统的处理能力。典型事件构成了超过某一阈值的参数值变化。通过采用告警机制，可以（通常）实时记录并报告这些变化。回想一下先前对计算机架构和操作系统的讨论，这种情况与设备引发的 CPU 中断标志非常类似。实际上，正如操作系统需要提供中断处理程序一样，网络管理系统也需要提供正确的操作流程。需要注意的是，为了检测变化情况（以及对变化做出反应），需要调用配置管理机制。

OSI 网络管理框架的第 3 项内容是性能管理。这也依赖于配置管理机制来实时测量网络资源的使用情况。这项活动的长期部分是容量规划。显而易见，当给定资源变得不堪重负从而影响到网络整体性能时，使用大型资源来替代现有资源可能要花更多时间（但确定哪些资源有助于缓解瓶颈是一个非常复杂的问题）。由于更换或补充设备通常成本昂贵，因而高效的容量规划可以节省大笔资金。

OSI 网络管理框架的第 4 项内容是身份和访问管理，这将在本章最后一节进行描述。简而言之，访问管理的任务是确保每次学习网络任何信息的尝试（或改变网络中的任何内容）能够实现，且只有在决定对尝试实体正确授权后，才可启动尝试进程。通常情况下，需要记录访问关键数据的尝试，否则需要通过故障管理机制进行处理。

OSI 网络管理框架的第 5 项内容（也是最后一项内容）是审计管理。这涉及与资源使用收费有关的一系列活动。在包括若干个机构的企业网络中，这可能意味着需要确定每个机构应当支付的整体通信费用比例。在运营商网络中，确定收入的主要依据是活动。

虽然 OSI 网络管理框架已经相当清晰（对于云来说，这一框架并未发生变化），但事实证明网络管理标准的开发是相当不稳定的。几个标准化组织并行开展竞争性活动，且迄今为止仍然没有结果。

从历史的角度来看，上述 5 项内容以不同次序拼读出来——故障、配置、审计、性能和安全性（Fault，Configuration，Accounting，Performance and Security，FCAPS），因而首字母缩略词为 FCAPS，这构成了 ISO 工作的基础，后期与 ITU - T 共同开展标准化工作。同时，根据同一模型，IETF 正在开发自己的协议系列。下面将简要介绍他们的协议，但是仍从共同的基本模型开始，如图 7.7 所示。

每台被管理设备都与管理信息库（Management Information Base，MIB）相关联，管理信息库实际上定义了配置参数。管理系统可以请求（获取）参数值和改变（设置）参数此值。哪些参数值可以从外部进行更改（如果需要对参数值进行修改，则需要明确由谁进行修改）是 MIB 规范的一部分。还有与参数有关的其他功能，如学习在 MIB 中定义了哪些参数，以及定义被管理对象的许多细微差别。告警

㊀ 1991 年的出版物[14]描述了法国电信新网络运营中心 Netminder™ 流量管理系统的部署情况。

㊁ 在某些运营支撑系统（OSS）中，其早期版本依赖于转椅和人工输入，但随后 OSS 升级到能够直接从交换机处接收该输入，而输入的形式保持不变（以确保接口相同，防止对 OSS 代码进行任何更改）。对交换机进行编程，达到假装是人类以 ASCII 文本来响应 OSS 提示的效果。

㊂ 考虑一家公司拥有多个局域网（LAN）并从电信运营商租赁线路来实现局域网互连的情况。公司的网络管理系统可以改变用于控制企业自身设备的参数值，但它只能读取用于控制运营商设备的参数值。

或陷阱消息是来自设备的通知，只能作为承载管理系统的机器处的中断事件进行处理。

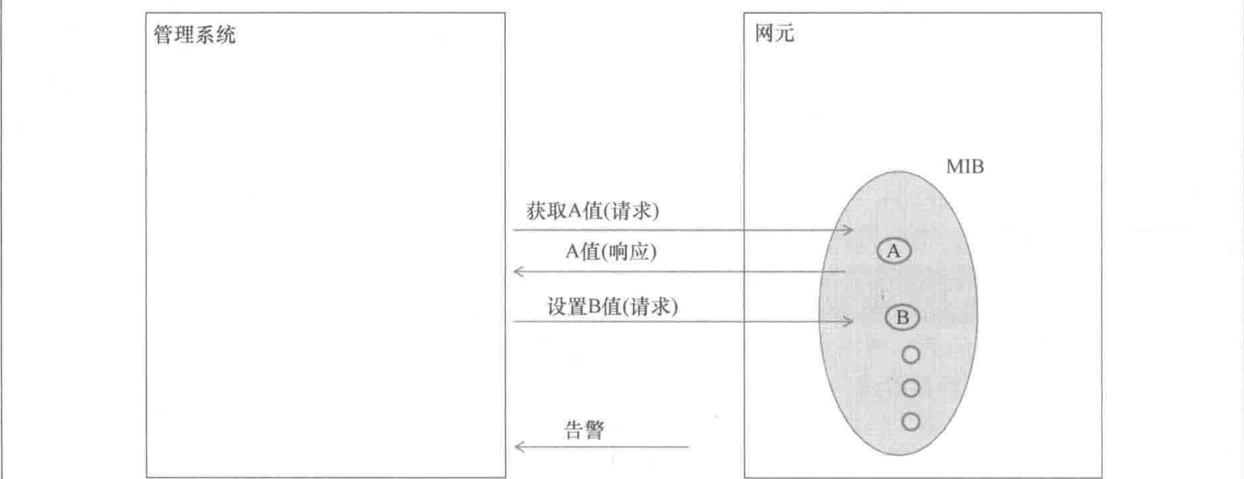


图 7.7 网络管理基本模型

国际电信联盟（International Telecommunications Union，ITU）已经联合 ISO/IEC JTC 1 共同开发了本章参考文献［15］中定义的通用管理信息协议（Common Management Information Protocol，CMIP）和 X.700 系列中的其他 ITU-T 建议。由于 CMIP 正在使用互联网中不可用的 OSI 应用层服务［如 OSI 远程操作服务元素（Remote Operations Service Element，ROSE）］，而 IETF 决定继续使用自己的协议，这样在网络管理标准开发方面开始出现分歧。

基于通用管理信息协议（CMIP）和其他模块，ITU-T 已经提出了一组称为电信管理网（Telecommunications Management Network，TMN）的规范（M.3000 系列），而 IETF 已经开发出所谓的简单网络管理协议（SNMP），目前已经演进到第 3 版——SNMPv3。企业 IT 部门已经专门部署了 SNMP，而将 TMN 部署在电话网络（特别是在 WorldCom）中，本章参考文献［16］对此进行了报道。这种差异是相当不幸的，因为它进一步扩大了电话和 IT 之间的差异——网络管理标准化工作应该消除的差异。

SNMP STD 62 标准于 2002 年完成，它反映了 SNMP 十多年的开发成果。到 2003 年，当 IETF 互联网架构委员会举办研讨会㊀时，SNMP 已经得到广泛部署，且大多数 IP 设备上已经拥有了一些管理信息库（MIB）。因此，业界已经获得了足够的操作经验来理解技术的局限性。技术局限性是 SNMP 主要用于处理网络管理的设备监控问题（与配置问题截然相反）。

应该强调的是，设备监控过去是（现在仍是）重要功能，因为它提供了物理设备（如服务器主板或简单风扇）状态的通知消息（“陷阱”）。知道硬件正常工作并尽早发现故障是操作规则的基础。在现代数据中心，这种 SNMP 陷阱被馈送到专门监控系统（如 Nagios㊁）中，并用作基于开发运维（devops）方法的现代解决方案的一部分。

虽然使用 SNMP 来配置设备并不是闻所未闻的（毕竟协议明确支持通过 SET 方法来更改设备参数），但许多标准的 MIB 模块仍缺乏可写对象。使用 SNMP，识别配置对象并不容易，且根据 RFC 3535 描述的内容，命名系统本身似乎是恢复到重新配置系统的先前配置。但是，对网络运营商来说，即使所有 MIB 都是完美的，SNMP 的水平也太低——他们对以鸟瞰图方式开发应用构建块所做的工作过少而感到遗憾。

随着 SNMP 软件开始达到其性能限制，这种构建块的开发可能帮助不大。例如，事实证明，检索路由表的速度非常慢。另一组问题是由于保持协议简单（这正是 SNMP 中 S 所表达的含义）的目标引起的。果然，协议相当简单（与 CMIP 相比），但却将复杂性留给开发人员处理。目前，网络管理应用本该检查 SNMP 事务的状态㊂，并准备将设备回滚到一致的状态。设计这种应用需要开发人员具有分布

㊀ 研讨会的结果已记录在文档 RFC 3535 中。

㊁ Nagios 是一个开源软件项目。

㊂ 在这种情境中，选择 UDP 作为传输协议使得问题变得更为糟糕，正如杰夫·休斯顿（G. Houston）在 2002 年所撰写的文章中指出的那样。



式处理方面的丰富经验，甚至对于专家来说，这也绝对不是一项简单的任务。这是会让人对实现网络管理应用更“易于”开发的计划产生误解的（即由来自一些基本组成部分的非程序员拼凑而成）。更一般地说，正如 RFC 3535 所描述的那样，“面向的世界任务视图和由 SNMP 提供的世界的数据中心视图之间语义方面存在着不匹配。从面向任务视图映射到数据中心视图通常需要一些有实际意义的代码。”

“简单性”导致规范不足，这将严重阻碍互操作性：

“几个标准化 MIB 模块缺乏对高级规程的描述。通过读取 MIB 模块，通常无法弄明白某些高级任务是如何完成的，这会导致实现同一目标可以采用几种不同方式，从而增加成本并严重阻碍互操作性。”

SMNP 在配置管理方面无效的一部分问题正是网络管理员（通常假定为“智能”）处理“哑设备”所使用的模型。最初，设备（调制解调器就是很好的例子）确实不是可编程的，但到了 20 世纪 90 年代末，情况发生了巨大变化。要理解不同之处，可以考虑家庭网络概念所发生的事情，家庭网络从庞大的调制解调器（将计算机连接到电话线）演变为以太网 LAN 集线器（尽管普通家庭中相对较为少见），然后演进到当前拥有内置防火墙和 NAT 盒的 Wi - Fi 基站路由器。除了明显差异之外，还有相当微妙的一点：所有这些设备引入的复杂性需要根据具体的策略进行配置更改。

有关 SNMP 的更多详细信息，推荐读者参阅下一章。

7.2.2　基于策略的管理 ★★★

IETF 开始逐步解决问题，并严格基于具体需求开展。首要需求是用于支持 QoS 的策略配置。这里，设备（通常是路由器）是智能的：其配置需要根据用户需求而不断变化（以响应用户需求），且管理系统需要将变化情况从本地副本传输到设备中。这里，模型引入了新挑战——需要在网络管理器和设备之间保持同步状态。另一个挑战来自于负责管理同一设备的两个或多个网络管理员之间存在的潜在干扰[㊀]。这种情况引入了矛盾变化可能导致设备损坏的隐患。

此时，就需要一种基于策略的管理。虽然设备可能必须根据用户请求发生变化，但是仅仅基于用户请求（即用户请求什么就提供什么）来分配网络资源几乎是不可接受的。网络提供商希望拥有一种能够基于一组策略规则来赋予资源的机制。关于是否授予资源的决定需要考虑到用户和账户信息、所请求的服务以及网络本身的信息。

将 SNMP 应用于此目的并不简单，因而 IETF 开发出一种新协议，用于网元与策略决定点（Policy Decision Point，PDP）之间的通信，而 PDP 也是基于策略做出决定的地方。将该协议称为公共开放策略服务（Common Open Policy Service，COPS），本书将在附录 A 中对该协议进行介绍。

重要的是，公共开放策略服务极大影响了 2004 年来 ETSI（European Telecommunications Standards Institute，欧洲电信标准化协会）和 ITU - T 开发的下一代网络（NGN）标准。NGN 的主要特征是：①端到端分组传输普遍采用 IP；②驱动有线和无线技术之间的融合[㊁]。

与针对特定应用进行优化的专用网络相反，人们将 NGN 设想为可以满足大量应用性能需求和安全性需求的通用多业务网络。为此，服务控制与传输、分配机制分离，并向应用提供（通常以实时或近实时形式）网络资源。

出现了支持所谓三重播放业务的一系列具体应用，包括 IP 语音（Voice over IP，VoIP）、IP 电视（IP Television，IPTV）和互联网接入。这些应用过去需要（且将来仍然需要）特殊的 QoS 处理。

正如在第 4 章中所看到的那样，应用性能需求可由 4 个关键参数：带宽、丢包、时延和抖动（即时延变化）来表征，这些都决定着服务质量。总体而言，三重播放业务需求在 QoS 方面是不同的。例如，一些流行的数据应用（如电子邮件和 Web 访问）需要低带宽到中等带宽，且在时延和抖动方面要求是相当宽松的。相比之下，视频点播（Video on Demand，VoD）流对时延的要求放宽，但是它们确实需要高带宽，且不能容忍丢包过多或抖动。然而，VoIP 可以容忍一些丢包，需要比 VoD 低得多的带宽，但它既不能承受长时延，又不能容忍抖动。

㊀ 在杰夫·休斯顿（G. Houston）的文章中对这两种挑战进行了解释，前面的脚注中提到了这一点。

㊁ 实际上，尽管无线网络中通信路径的一小部分（即手机和无线基站之间的路径）是无线的，其余部分的路径大部分是“有线”的，可是历史上电信运营商仍坚持两种网络和服务分开提供。2005 年似乎是推动融合的关键一年。本章参考文献［17］是一部关于这个主题的综合性专著。

除了与 QoS 相关的资源之外，网络通常需要向端点和进程赋予其他资源（如 IP 地址或与服务相关的端口号）。回顾一下第 5 章的内容，这一特定需求来自于 NAT LSNAT（Load Sharing using IP Network Address Translation，使用 IP 网络地址转换的负载共享）部署，采用这种部署模式可以隐藏内部网络拓扑。

NGN 特有的结构使得这些多样化且已经很复杂的任务变得更加复杂，因为 NGN 结合了多种网络类型，包括异步传输模式（Asynchronous Transfer Mode，ATM）、数字用户线（Digital Subscriber Line，DSL）、以太网以及固定和移动无线接入网络。

履行这一复杂职责的关键是一种基于策略的动态资源管理框架，本章参考文献［18］将其称为资源和访问控制功能（Resource and Admission Control Function，RACF）（读者可参阅本章参考文献［19］中发布的 ITU – T 标准）。需要强调的一点是，RACF 已经实现了与 OSS 的协同实时处理。RACF 拥有两项功能，且该协议综合了两组构建块。

即使 RACF 直接受到 COPS 的影响，其框架也依赖于除 COPS 之外的一些 IETF 协议㊀。从诞生之日起，第 3 代合作伙伴（3GPP 和 3GPP2）一直在跟踪和影响支持 IP 多媒体子系统（IP Multimedia Subsystem，IMS）的 IETF 构建块的开发工作㊁。

虽然第 3 代合作伙伴关注的重点是无线运营商的需求，但 ETSI 电信和互联网融合业务及高级网络协议（Telecommunication and Internet Converged Services and Protocols for Advanced Networks，TISPAN）小组于 2003 年启动了一个涉及固定接入的项目。该项目中的资源管理方法体现在其资源和访问控制子系统（Resource and Admission Control Subsystem，RACS）中，该标准发布于本章参考文献［21］中。

应当再次强调，网络地址端口转换以及 NAT 穿越的控制需求是 ETSI 标准化工作的重要推动力。当服务提供商开始部署 VoIP 时，他们发现了我们目前非常熟悉的、由位于 NAT 设备后的终端用户导致的并发症（大多数宽带接入用户也会遇到类似情况）。采用支持托管 NAT 穿越的会话边界控制器能够有效避免此类问题的出现。然而，独立会话边界控制器在整个 IMS 方法中表现并不太理想。相比之下，在与 IMS 会话控制接口的策略决策功能的控制下，作为策略执行的一部分，RACS 模型支持 NAPT 和托管 NAT 穿透。

2004 年，ITU – T 开始着手进行 RACF 的工作，目的是保持服务和传输的分离，同时支持应用驱动的、基于策略的动态端到端 QoS 和资源控制能力（特别是资源预留）、访问控制和门控、NAPT、网络域内和网络边界的托管 NAT 穿透。从一开始，RACF 的适用范围就包括各种类型的接入网和核心网。为此，通过混搭（而不是取代）公共框架内的现有标准化成果，RACF 是创建灵活端到端资源管理解决方案的首次尝试。

可以看出，COPS 已经解决了基于策略的管理问题，但并没有解决高效管理配置问题。回到 RFC 3535（读者可能记得，有关 2002 年 IAB 研讨会的报告），运营商普遍的投诉是缺乏全面一致的配置规程。无论是 COPS 还是日益增长的 IETF MIB 集都于事无补。

因此，专题讨论会的目标之一是确定如何重新聚焦 IETF 资源㊂。研讨会提出 8 条建议，既有积极

㊀ 这些协议包括远程认证拨入用户服务（Remote Authentication Dial – in User Service，RADIUS）及其后续协议：Diameter 和媒体网关控制（Media Gateway Control，MEGACO）。RADIUS 是一种用于远程拨入应用中的早期认证和授权协议，但易于扩展到其他访问机制。最初，Diameter 是作为 RADIUS 的改进版出现的，但目前已演变成大量需要认证、授权和计费功能的应用所广泛采用的通用对等协议，这也解释了它成为资源控制必不可少的候选方案的原因。MEGACO 协议由 IETF MEGACO 小组与 ITU – T 第 16 研究组联合开发，且它在 ITU – T 项目名称是 H. 248。最初，人们开发 MEGACO/H. 248 的目的是用于管理电路和分组网络之间的媒体网关，后来将其进行扩展以支持通用分组到分组的边界网关。需要注意的是，与 Diameter 或 RADIUS 不同，MEGACO/H. 248 不是对等协议，它是一种客户端/服务器或（使用自己的术语）主从协议。在某些情况下，此功能限制了它在策略决定点（PDP）到策略执行点（Policy Enforcement Point，PEP）接口的有效性。

㊁ IMS 是由 3GPP 开发的，从 1998 年开始，朝着 IP 和无线融合迈出了决定性一步。本章参考文献［20］对 IMS 架构和标准进行了描述。在 21 世纪的前 10 年，3GPP 已经在基于业务的本地策略（Service – Based Local Policy，SBLP）中采用了这些构建块，而 SBLP 正在向策略与计费控制（Policy and Charging Control，PCC）机制演变。人们将 3GPP2 模型称为基于业务的承载控制。尽管相应接口指定的协议有所不同，可是它们在概念层面上非常相似。

㊂ RFC 3535 使用严谨的术语对此进行了描述：“在这些会议召开期间，几家运营商表达了自己的看法，即 IETF 在标准开发方面并没有真正考虑他们的需求，特别是在配置管理方面。这自然会导致 IETF 是否应重新聚焦资源，以及未来在运营和管理领域开展哪些战略性活动等问题。

的（需要重点关注哪些活动），又有消极的否定（哪些活动需要停止）。其中，“协议开发人员和运营商达成强烈共识”的唯一获得协议开发人员和运营商的积极建议是“IETF 将资源集中于配置管理机制的标准化”。另两条运营商支持力度明显大于协议开发者的建议是：将资源用在“基于 XML 的设备配置和管理技术的开发和标准化”，而不是用在基于 HTTP 上 HTML 的配置管理[1]。

值得赞扬的是，IETF 反应相当敏捷，并给出决定性回应。2003 年，NETCONF 工作组成立，3 年后发布第 1 版 NETCONF 协议。在接下来的两年中，人们将通知以及几个经典的分布式处理和安全机制增加到协议中，且协议在后面 8 年中不断演变。基本 NETCONF 协议的现有版本——RFC 6241 是 2011 年 6 月发布的。其扩展内容已发布在单独的 RFC 文档中。本书将在下一章中对 NETCONF 协议进行详细介绍。

同时，业界已经开发出若干种配置管理工具，这些工具在当今的云中得到了广泛应用。在本节的剩余部分，将回顾两个众所周知的实例：Chef 公司（以前的 Opscode 公司）开发的 Chef 配置工具和 Puppet 实验室开发的 Puppet 配置工具。

使用 Chef 配置工具，管理员使用所谓的配方来描述分布式系统（可能包括 Web 服务器、负载均衡器和后端数据库）结构。这些配方描述了如何部署、配置和管理结构中的实体，而它们存储在 Chef 服务器上。Chef 客户端安装在各个节点（它可能是虚拟机）上。Chef 客户端的工作是通过检查安装在 Chef 服务器上最新配方的符合度，并根据需要自动更新软件，来确保每个节点上的软件处于最新状态。在编写本书时，该公司在其网站上提供了作为学习工具的免费实验（甚至是受限的免费软件分发），我们将其强烈推荐给感兴趣的读者。

类似地，Puppet 能够自动执行配置任务，因为它也是基于客户端/服务器模型的。它与 Chef 的主要区别在于规范方法。Puppet 规范（使用自己的 DSL）是声明式的——它指定从属关系，且客户端确保遵循这些规范。相比之下，Chef 规范是程序化的，它是使用 Ruby 语言编写的。与 Chef 一样，Puppet 也是开源可用的。本章参考文献［22］是一篇鞭辟入里的文章，它对两者进行了全面比较。

7.3 云内部的业务流程与管理

这里准备将本章和本书其他部分涉及的难题拼凑在一起。前面已经介绍过数据中心和网络互连物理单元的管理元素，没有涉及的内容是云服务生命周期的管理问题。

除了许多技术问题（如图像的创建和引导），这里开始讨论业务问题。分布式管理任务组（Distributed Management Task Force，DMTF）[2]组织已经对该主题进行了深入介绍，因而将使用 DMTF 白皮书——《管理云的架构》中描述的定义和概念。下一节将介绍 OpenStack 中的业务流程和管理。

这里，需要强调的是，业务流程可以在各个层次上实现。在本章中，用音乐模拟开始，并以音乐模拟结束。最终，在乐团里，每件乐器都需要发挥自己的作用。这些作用可以在音乐家“群”（如第一小提琴组或第二小提琴组）之间共享，但最终将各个部分合并成为节，然后组合成单个乐谱——乐队指挥需要用到的整个乐谱。

在极端（或有些退化）的情况下，需要由自己的管理员对云中的每台虚拟机进行手动配置、监视和重定位等。此项任务可以使用工具（如 Chef 或 Puppet）[3]来实现自动化。这是虚拟机（“乐器”）。拥有自身“乐谱”的地方。在新的层面上，整个基础设施（虚拟机和网络组件将其进行互连）可以根据统一的“乐谱”进行安排，这就是 OpenStack 实例所要说明的问题。但事情远不止此！在顶层，可以将商业策略与基础设施规范结合起来编写“乐谱”。这可以通过使用 OASIS 标准——云应用拓扑和业务流程规范（Topology and Orchestration Specification for Cloud Applications，TOSCA）来实现，本书将在附录 A 中对该规范进行回顾。

7.3.1 云服务的生命周期 ★★★

这里涉及的 3 个实体是云服务提供商、云服务开发人员和云服务消费者。

[1] 正如事情发展的那样（稍后会回到这个话题），NETCONF 战斗是 REST 阵营失利的战斗之一。当时，RPC 阵营赢了，但它在 2009 年左右开始系统性溃败，此时 REST 阵营重新集结。

[2] DMTF 成立于 1992 年，其宗旨是“将 IT 界联合起来，共同开展系统管理标准制定、验证、推广和采纳”。

[3] 这种自动化甚至可以扩展到相同虚拟机的集群。

假设云服务开发人员需要创建一种（典型的）Web 服务基础设施，如 3 台相同服务器、1 台负载均衡器和 1 个后端数据库。编写一种向云服务提供商发出个人请求的程序，来创建所有实例和网络，它在多个方面存在问题。

首先，假设已经成功创建了负载均衡器和 2 台服务器的实例，但是在为第 3 台服务器创建的虚拟机失败。用户程序应该做些什么？删除所有其他实例并重新启动不是一种有效的做法，原因如下：从服务开发人员的角度来看，这将导致程序严重复杂化（程序本应是相当简单的）。从服务提供商的角度来看，这将导致最初分配资源的浪费，然后释放从未使用的资源。

其次，假设所有实例都已经创建，则服务提供商需要支持弹性。问题是：①如何规定弹性；②弹性将产生何种影响。假设 3 台服务器中的每台都达到 CPU 利用率的阈值。这样，简单的解决方案是创建另一个实例（一旦突发活动结束即可进行删除），但是如何自动完成所有这些实例的创建？为此，也许可能不需要 3 个实例，而只需首先创建两个实例。

业界采用的解决方案是以更一般的术语来定义服务（将用实例来说明），这样服务创建就是由服务提供商执行的一种原子操作——这是业务流程首先引人注意的地方。一旦部署了服务，业务流程管理器自己就会添加和删除服务定义中指定的实例（或其他资源）。

因此，图 7.8 对工作流进行了描述。服务开发人员定义了模板中的服务，它还指定了服务接口。模板（有时业界也称为配方）指定了各种资源：虚拟机（VM）图像、连通性定义、存储配置等。

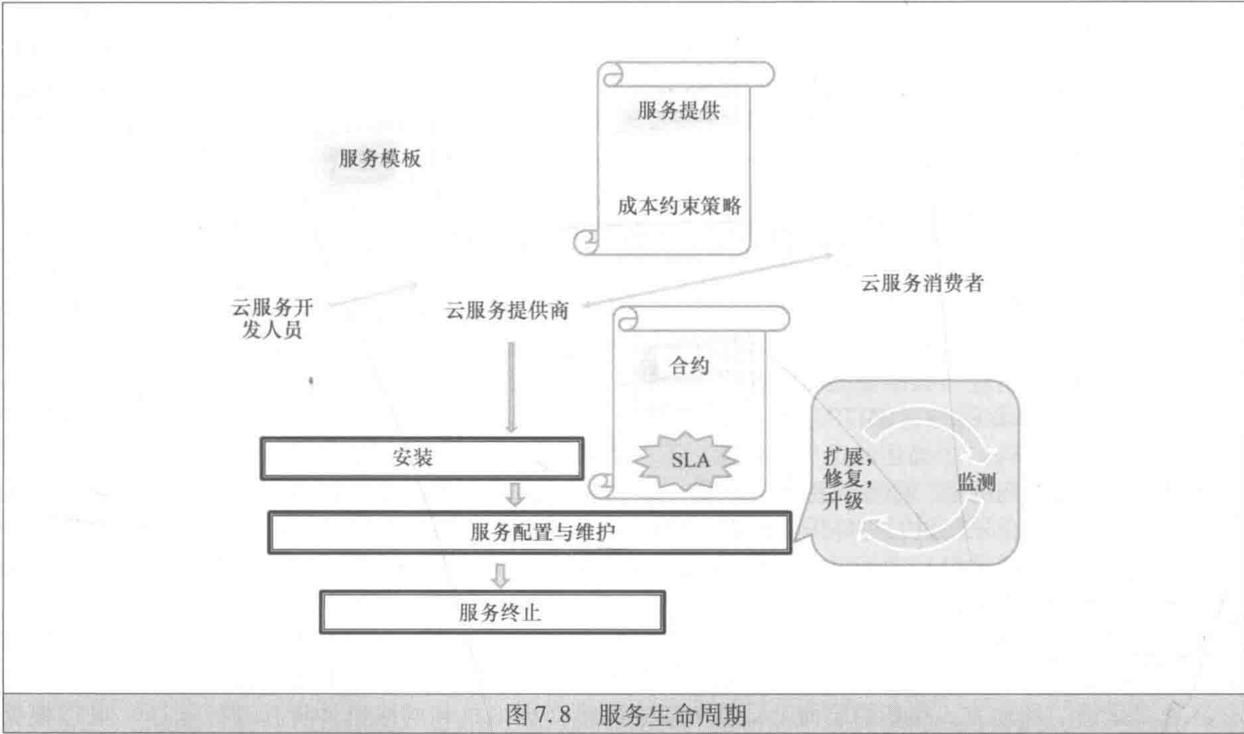


图 7.8　服务生命周期

服务提供商通过使用约束条件、成本、策略和 SLA 来扩充模板，从而为服务消费者提供服务。在接受服务时，消费者和提供商签订了一项合约，包含 SLA 以及一组具体可衡量的 SLA 问题，称其为服务水平目标（Service Level Objective，SLO）[⊖]。

此时，提供商可以修改模板来满足合约要求。然后，提供商基于模板来部署（或提供）服务实例。服务提供涉及履行合约所需的资源。

一旦完成部署，就需要对服务进行维护，直到合约终止，因而服务结束，支撑服务的承诺资源被重新部署。从业务流程的角度来看，服务维护的一个重要组成部分是监控。这里，相关事件将被自动收集，并据此采取行动，以便在发生故障时增加或减少容量以及修复服务。同样，在这一阶段升级也会自动完成。自动扩展和自动修复功能是业务流程的两大功能。

⊖ SLO 可用于确定审计中的合规性。这些测量值有望被记录下来。



正如所看到的，该模型意味着业务目标和接口定义能以某种形式来表示（即编码）。电信管理论坛（Telemanagement Forum，TMF）[一]开发了应用于此目标的形式语言结构。DMTF/TMF 联合白皮书——《通信服务提供商的云管理》探讨了 DMTF 与 TMF 之间的协同效应。

从进入角色[二]开始。这里，服务开发人员需要指定哪些应用在哪些虚拟机上运行，业务流程管理器需要处理哪些事件（以及这种事件发生时究竟要做什么）和需要收集何种信息。

应用配方（或模板）描述了应用所需的服务，每种服务都可进一步定义为一组服务实例（运行于单独的虚拟机上）。这些都是作为文件描述符提供的。每个指定了实例数量、软硬件要求、生命周期事件及其“中断处理程序”的配方对服务进行了进一步规定，它们是对应于相应脚本的指针。为了支持网络和运营管理，配方还可以指定用于监控和配置管理的探测器。除了服务开发人员可用的预定义探测器之外，后者还可以插入独立脚本。云管理和业务流程存在的一个问题是，必须最优利用履行客户义务所需的云服务提供商资源（主要是成本方面）。这里，优化是一项复杂的任务，因为存在诸多限制条件，包括遵守客户策略和各种规定。

另一个问题是为客户提供业务流程工具，以便客户可以控制自己的基础设施。最终，提供商可能会与客户分享一些自己的业务流程工具。由于业务流程涉及与业务活动的互通，因而采用工作流支持工具正在成为预期功能。例如，VMware® vCenter™ Orchestrator™提供了一种预构建工作流库以及用于设计定制工作流的工具。可以使用基于 JavaScript 的脚本引擎来创建新的工作流块。策略引擎根据所定义的策略来启动合适的工作流来响应外部事件。

另一个重要的实例（在某种程度上，可以看作是业务流程管理器的基准）是 Amazon AWS CloudFormation 服务，它提供了一种管理 AWS 基础设施部署的机制。

正如将要看到的那样，OpenStack 的业务流程管理器——Heat 采用了 AWS CloudFormation 的术语和模板格式，而在其早期的业务流程中，围绕 AWS CloudFormation 为用户提供的相同工具和接口之间的互通开展了大量工作。采用 AWS CloudFormation，所有资源和从属关系都在模板文件中进行声明。每种模板均定义了与服务相关的资源集合，以及它们之间的从属关系。将实际上代表某种基础设施的集合称为一个堆栈[三]。思路是将给定堆栈中的大量资源看作是单个实体，它可以通过一条 create 或 delete 命令来创建（或删除）。

同时，当对堆栈模板进行更新时，堆栈也被（自动）更新。此外，一旦指定了模板，则可以对整个堆栈进行复制，甚至将其移动到不同的数据中心，甚至是不同的云。

图 7.9 对这一概念进行了描述。这里，模板定义了之前讨论的基础设施：负载均衡器，用于在 3 台相同服务器之间分配流量。为使服务看起来更真实一些，还添加了后端数据库。这里涉及两个网络：一个网络在负载均衡器和服务器之间共享，另一个网络在服务器和数据库之间共享。使用一系列｛create，delete，create｝操作，首先在云中创建整个基础设施，然后在另一个云中进行复制。当然，这里假设云提供商都支持相同的模板，正如将在下一节中看到的那样，OpenStack 项目已经通过创建标准以及实现它的软件达到了预期目标。

根据本节开头提出的思路，强调在创建（或删除）堆栈时，模板中指定的所有资源同时被实例化（或删除）。在堆栈的生命周期中，资源之间声明的从属关系将会自动得到维护。

首先，亚马逊在世界各地部署了拥有已知 URL（Uniform Resoure Locator，统一资源定位符）的 CloudFormation 端点。参考本地地理端点能够减少时延。正如将看到的，一些功能性指标取决于端点的选择。

该模板是使用 JavaScript 对象表示法（JavaScript Object Notation，JSON）格式编写的[四]。除了版本号和描述字段之外，它还包含以下字段：资源、参数、映射、条件和输出。在图 7.10 的帮助下，本书依次对它们进行回顾。

[一] TMF 创建于 1988 年，其名称为 OSI/网络管理论坛，旨在协助其成员解读 OSI 标准，以开发可互操作的网络管理解决方案。1998 年，该组织更名为电信管理论坛，且从此一直在快速发展。

[二] 这一术语（业界非常流行）是在 Jelle Frank van der Zwet 的白皮书中定义的：“在迁移到云环境的情景下，‘进入角色’是指将应用、数据或二者部署到所选择的云基础设施（公共云、私有云或混合云）上。”

[三] 这一术语已被广泛采用，特别是 OpenStack 组织甚至影响到其名称。

[四] IETF 已经对该格式进行了标准化。

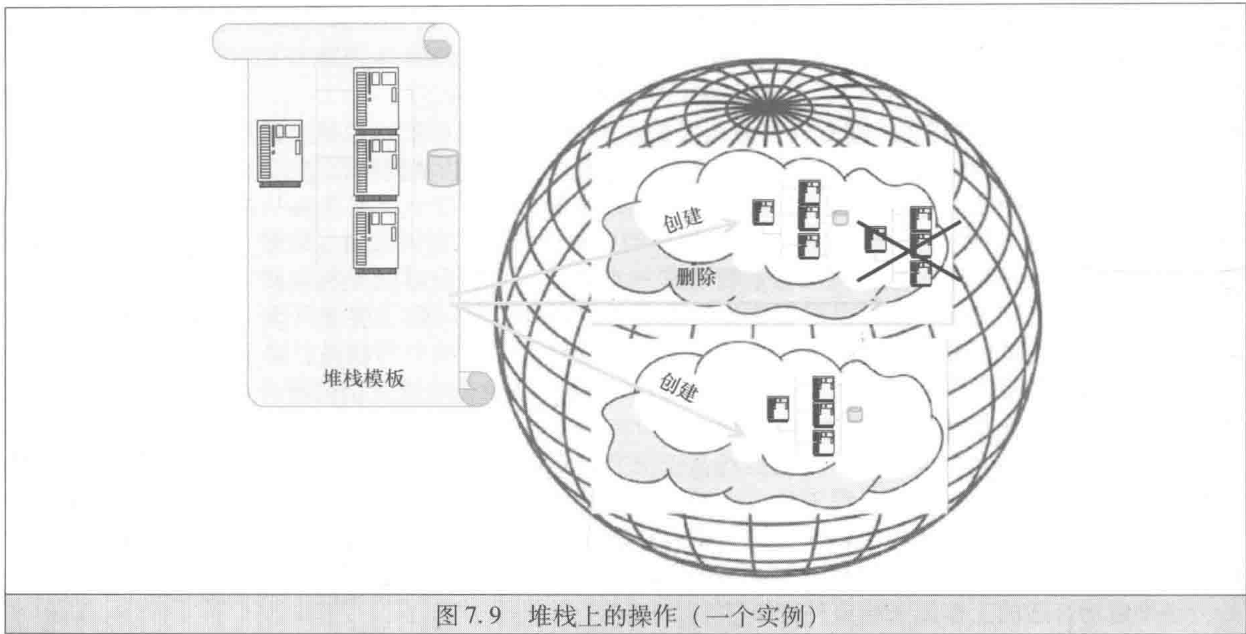


图 7.9　堆栈上的操作（一个实例）

```
{
  "AWSTemplateFormatVersion" : <date>
  "Description" : <string>

 "Resources" :
   { <resource-name/type/properties list>
   },                                        强制性的

 "Parameters" :
   { <parameter list>
   },

  "Mappings" :
    { <mapping/key/value list>
    },

  "Conditions":
    { <name/intrinsic function/arguments list>
    },

 "Outputs" :
   {
   }
}
```

图 7.10　AWS CloudFormation 模板

术语资源是指虚拟机实例或任何其他 AWS 预定义对象（如安全组或自动扩展组——很快将看到具体实例）。我们为每种资源分配一个资源名称，且在模板内资源名称是唯一的。

资源类型是资源规范的另一部分。此外，还可以声明与资源相关的一组资源属性，每项声明均采用名称/键值对的形式。

属性值可能只在运行时才是已知的，因而模板语法允许使用内部函数而不是静态值。资源项是唯一的强制性项，其余项均是可选项。

参数只是一个名称字符串，其规范可能列出限制条件，这些条件会影响到参数的取值。

映射能够实现参数值自动分配。人们可以定义参数值的一个子集，并将其与一个键相关联。键的典型实例是地区名。所有与区域有关的键值（如当前时间或当地法规）均被自动分配给相应参数。

条件只是一种程序工具。它们是用于将参数值相互比较或与常量进行比较的布尔函数。如果比较的结果为正，则创建资源。当创建或更新堆栈（直到那时）时，需要对所有条件进行评估。

输出是声明专门支持反馈机制的参数。最终用户可以通过 describe - stack 命令来查询任何输出值。此外，还可以使用条件来引导键值分配。

回到早期的 Web 服务实例，可以看到如何构建模板来支持自动扩展——支持弹性的、业务流程管理器提供的服务。特别是在 AWS 中，自动扩展功能可以根据用户定义的策略以及运行时的特性（如通过监视来衡量应用的“运行状况”）来启动或终止实例。通过改变实例的计算能力，可以实现垂直扩展，或通过改变负载均衡实例的数量来实现水平扩展。特别是能够展示云环境独特经济优势的水平扩展：在物理部署中，需要保持有额外的服务器处于待机状态，以期增加负载（或所有服务器的实际负载平衡，而服务器未得到充分使用），但在云环境中，当需求达到指定阈值时，能以实时形式部署一个额外服务器实例。相反，当需求下降明显时，可以关闭多余的实例。因此，只有在需要资源时，额外资源的支出才会发生。

用于操作涉及一组 Web 服务器环境的模板[1]将在资源报头下指定一组 AWS:: AutoScaling:: AutoScalingGroup，它列出了可用性区域、配置名称（另一个资源，指向要启动实例的图像），以及组的最小尺寸和最大尺寸等属性。

如果希望向操作人员通知事件（一项有趣的特征），则通知主题也可以指定为类型为 AWS:: SNS:: Topic 的资源，该资源是指适当的资源——端点（操作人员的电子邮件），并指定协议（“电子邮件”）。在这种情况下，公共组规范也将列出特定的通知消息字符串（如“实例启动”“实例终止”或“错误”，后者也提供适当的错误代码）。

接下来，可以使用资源类型 AWS:: AutoScaling:: ScalingPolicy 来规定放大和缩小策略。也可以将触发放大（或缩小）的实际告警事件指定为资源，如“类型”：AWS:: CloudWatch:: Alarm。例如，如果放大需求是 CPU 猝发利用率超过 80% 达 5min，则扩展告警的属性将包括使用 WS / EC2 命名空间 MetricName: CPUUtlilization、Period: 300 和 Threshold: 90。AlarmActions 是指上面定义的扩展策略名称。这里使用的内部函数是 ComparisonOperator，其值为 GreaterThanThreshold。

需要指定的另一种资源是负载均衡器本身，类型为 AWS:: ElasticLoadBalancing:: LoadBalancer，其属性包括要监听的端口号（假定使用的 Web 服务器已知）、实例端口号和协议（HTTP）。

最后但并非最不重要的是，必须创建用于描述 InstanceSecurityGroup 类型的实例安全组的资源。典型的用途是支持基于安全外壳（Secure Shell，SSH[2]）的访问仅限于前端——负载均衡器。

参数部分定义了上述结构：允许的实例类型、特定端口号、操作人员的电子邮件、SSH 访问的密钥对以及（CIDR）IP 地址模式。

映射部分提供参数值（在 AWS 中进行预定义），输出部分将列出唯一的输出——服务器提供的网站 URL。这可以使用内部函数来实现。当然，URL 的方案是 http，字符串的其余部分通过内部函数 GetAtt 获得，其中包含两个参数——资源部分中指定弹性负载均衡器资源的名称以及字符串 DNSName。这两个字符串可以使用内部函数 Join[3]和 1 个空分隔符级联起来。因此，如果弹性负载均衡器的名称是 MyLB，则输出部分如下所示：

```
"Outputs":
  {
   "URL":
      {Value":
          {"Fn::Join": ["",[http://",
              {"Fn::GetAtt": ["MyLB",
                "DNSName"]
          }]]
      }
  }
```

[1] 具有本节所述能力的超集的实际即用型样本模板。

[2] 本章后面，在回顾公钥密码学的情境中将讨论这一协议。

[3] Join [x, "d", y] 用于将字符串 x、分隔符“d”和字符串 y 级联起来。

正如前面已经提到的那样，详细描述 AWS CloudFormation 实例事出有因——因为它是一个基准。为此，OpenStack 中的业务流程管理器采纳了相同模板，这些将在下一节中进行介绍。当然，模板只是定义了需要做的事情，而如何做是另一回事。OpenStack 是一个开源项目，它允许我们了解云业务流程的内部工作原理，甚至可以参与相关软件的开发。

正如前面所说，业务流程编排可以在不同级别执行。本书将在下一节中讨论堆栈级业务流程的实现方案，并返回到业务流程编排这一主题，它包含附录 A TOSCA 讨论中的业务逻辑。

7.3.2 OpenStack 中的业务流程和管理 ★★★

首先，关于 OpenStack 本身我们简单介绍一下。用组织自身的话讲，其软件“...... 是一种能够控制整个数据中心大型计算、存储和网络资源池的云操作系统。在赋予用户通过 Web 接口提供资源的同时，所有这些资源通过为管理员提供控制功能的仪表板进行管理”。

该项目由 OpenStack 基金会有组织地提供支持。OpenStack 基金会的资金（至少是部分资金）来自公司赞助，但在其他方面 OpenStack 基金会吸引了成千上万的成员，且其个人会员是免费的。OpenStack 基金会的战略性管理是由董事会（Board of Directors，BoD）提供的。董事会代表着不同类型的成员。技术委员会（Technical Committee，TC）负责定义并指导 OpenStack 软件的技术方向。软件用户的支持和反馈由用户委员会负责。

实际上，在描述 OpenStack 组件时，本节是本书的一个高潮，因为它最终汇集了其他章节的材料。OpenStack 的软件组件完全对应于前几章中所研究的功能实体——存在着一种用于计算的组件（即对主机的管理，且主机能够提供由托管虚拟机共享的 CPU），它支持虚拟化；存在着一种负责联网的组件和一种负责存储的组件。负责与所有这些组件进行交互的是管理功能实体，尤其包括业务流程、身份和访问管理（这将在本章最后一节中进行讨论）等管理功能实体。需要注意的是，在调用这些“组件”时必须非常小心，因为这些组件都无法代表简单的架构实体，如机器或进程或库。正如将要提到的，一些组件将可执行图像、各种库和 Shell 脚本结合起来。

根据先前讨论的 API 术语，OpenStack 文档将用于实现 HTTP 服务器（通过 REST API 来访问）的组件部分称为服务。重要的是要理解，在物理主机上部署 OpenStack 软件是完全不同的一件事。总的来说，部署这些组件没有一成不变的方法。我们将提供具体的实例，需要注意的是部署问题几乎都可归结为确保与运营预算相当的可靠性。

首先，仔细观察组件。每一个组件都与负责其软件开发的单独项目相关。组件名称及其关联项目可以由 OpenStack 文档替代使用。

OpenStack 计算组件（是在名为 Nova 的项目中开发的）包含用于管理所有虚拟机生命周期的功能实体，因为涉及虚拟机的创建、调度和关闭。在计算中，控制器（云控制器、卷控制器和网络控制器）分别负责处理计算资源、块级存储资源和网络资源。

OpenStack 联网组件（在 Neutron 项目中开发）涉及为所有其他组件提供网络连接支持。OpenStack 管理指南将其称为“其他 OpenStack 服务的网络连接即服务”。该组件提供的服务支持网络连接和寻址，但重要的是，还存在一个可以插入其他软件的地方。目前，本地 Neutron 软件支持为所有 API 配置 TLS（Transport Layer Security，传输层安全）协议，并实现了负载均衡器即服务（Load Balancer as a Service，LBaaS）和防火墙即服务（Firewall as a Service，FWaaS）。

Neutron 还支持创建路由器，这些路由器是部署在节点上的虚拟机网关，而 Neutron L3 代理软件则运行于节点上。除此之外，路由器为浮动 IP 地址（属于云提供商的公共 IP 地址）执行 NAT。Neutron 设计的独特之处在于，这一地址不是通过 DHCP（Dynamic Host Configuration Protocol，动态主机配置协议）分配或静态设置的。关于这一点，客户操作系统不知道，因为传输到浮动 IP 地址的数据包是由 Neutron L3 代理专门进行处理的。这种安排提供了很大的灵活性，因为浮动（公有）和专用 IP 地址可以在任何网络接口上同时使用。

为了处理详细的网络管理信息，Neutron 支持插件。可以预见，存在着一个称为 Open Daylight 的开源 SDN（Software Defined Network，软件定义网络）项目（Linux 基金会的一部分）。推荐读者访问该项目网站，该网站提供了用于实现 SNMP 和 NETCONF 的插件的详细实例。显而易见，NETCONF 在 SDN 情境中意义重大。当然，还存在其他的实现方案，包括若干种专有实现方案。插件在后端运行。前端

REST API 支持包括创建和更新租户网络以及特定虚拟路由器等功能。

就存储而言，OpenStack 包含两个项目：Swift 和 Cinder。前者用于处理非结构化数据对象，而后者提供对持久性块存储的访问（这里再次预留有插入其他块存储软件的空间）。

服务组件（在 Glance 项目中开发）也与存储（相当专业的类型）相关。顾名思义，该服务负责处理存储和检索虚拟机图像的注册表。图像数据库的状态在 Glance 注册表中进行维护，而通过 Glance API 来调用服务。

认证和访问授权组件在 OpenStack Keystone 项目中执行，该项目涉及身份和访问管理。鉴于这一问题的独特之处，专门用了单独一节（本章最后一节）来讨论这些问题。

最后，存在着 3 种管理和业务流程编排组件。用户接口在“旧”CLI（Command Line Interface，命令行界面）表单和基于 Web 的门户网站中是可用的，OpenStack 信息板是作为 OpenStack Horizon 项目的一部分开发的。另两种组件是：①OpenStack Ceilometer 项目开发的遥测[㊀]，主要负责计量（通过监控来实现）；②OpenStack Heat 项目开发的业务流程[㊁]。为了解决这些问题，需要更加深入地了解 OpenStack 的架构，并通过一些部署实例进行说明。

在涉及这一话题时，必须牢记正版软件模块可以在任何地方运行，而 OpenStack 设计在定义高级软件接口（包括 REST API 和 RPC）方面已经迈出一大步，以确保管理活动相互影响的方式与硬件部署无关。这里，澄清“相互影响”这个词是非常重要的。根据上下文，它有两层含义：①子程序调用——它是活动中使用的编程结构；②传递消息——这是活动之间的交互方式[㊂]。要牢记的另一件重要事情是，为了确保可靠性，预计将在多台机器上复制数据和代码，因而这里的活动实际上可以被若干个相同过程所支持。当一个代码单元运行时，它运行于特定机器上，因而为了说明事件的关键序列，不存在复制的最小部署是非常有用的。一旦理解了这一点，则接下来要理解的是，与性能、可靠性和法规相关的要求不同，软件组件能够且可能需要不同的部署。

图 7.11 提供了第一个部署实例。我们将托管虚拟机的数据中心中的主机称为计算节点。因此，每个数据中心必须至少拥有一个计算节点。除了运行管理程序和托管客户虚拟机之外，计算节点还运行属于管理基础设施的各种应用。一些应用（将其称为代理）与其他组件（因而充当客户端的角色）发起交互；其他应用响应别处发起的通信（因而充当服务器的角色）。通常情况下，有些应用可能会根据具体情况充当客户端或服务器的角色。

就虚拟机管理程序而言，OpenStack 通过特定的计算驱动程序与几个主要的虚拟机管理程序进行交互，但交互程度有所不同。

实际上，计算代理正在创建和部署虚拟机。它充当调度器的服务器（将在控制器节点的情境中进行讨论），但它在处理中央资源数据库、图像节点和存储节点时充当客户端，它们分别保存了 Glance 图像注册表和各种存储类型（块或对象）。

存在于所有 3 种节点中的遥测代理收集了业务流程中使用的性能数据，本书将很快对这一问题进行讨论。

最后，控制器节点位于云管理的核心。首先，它包含全局资源数据库[㊃]，在介绍计算代理时曾经提及过。由于在实际的大规模部署中，这一数据库被复制，因而存在着一种称为 Nova Conductor 的前端，它主要负责处理计算代理接口。

调度器负责配置功能实体[㊄]。它需要决定创建新虚拟机的位置（即在哪个计算节点上创建），以及在哪个存储节点上创建新的块存储卷。前者使用 Nova 调度器，而后者使用 Cinder 调度器。

㊀ 在 OpenStack 软件的早期版本中，OpenStack Cloud Watch 项目（AWS Cloud Watch 服务类似的功能，它支持告警设置、指标收集和日志文件监控）中开发了指标收集和告警配置。目前，这一项目已经被 Ceilometer 取代。

㊁ 我们试图不追究 OpenStack 项目名称的词源，尽管这本身就是一项有趣的研究。当涉及业务流程编排项目时，其名字是相当合适的，绝对不是随机的。我们的文字处理器不能识别“测云仪”这个词，因而只有在网上查找该词的准确含义。根据美国国家科学数字图书馆的统计数据，测云仪是“使用激光或其他光源来确定云底高度的设备”。明白了吧！至于 Heat 项目，OpenStack 网站主动提供对项目名称的解释：“为什么叫‘Heat（热）’？因为它能使云升起！”

㊂ 甚至更高级的活动，这些活动是前面讨论的系统信息块（System Information Block，SIB）分布式执行的结果。

㊃ 在 OpenStack 中，选择的数据库软件是 MySQL——可以从同名开源项目处获得。

㊄ 在附录 A 中，专门用了一节的篇幅来介绍这一复杂而有趣的课题，需要通晓优化技术，因而被视为高级课题。

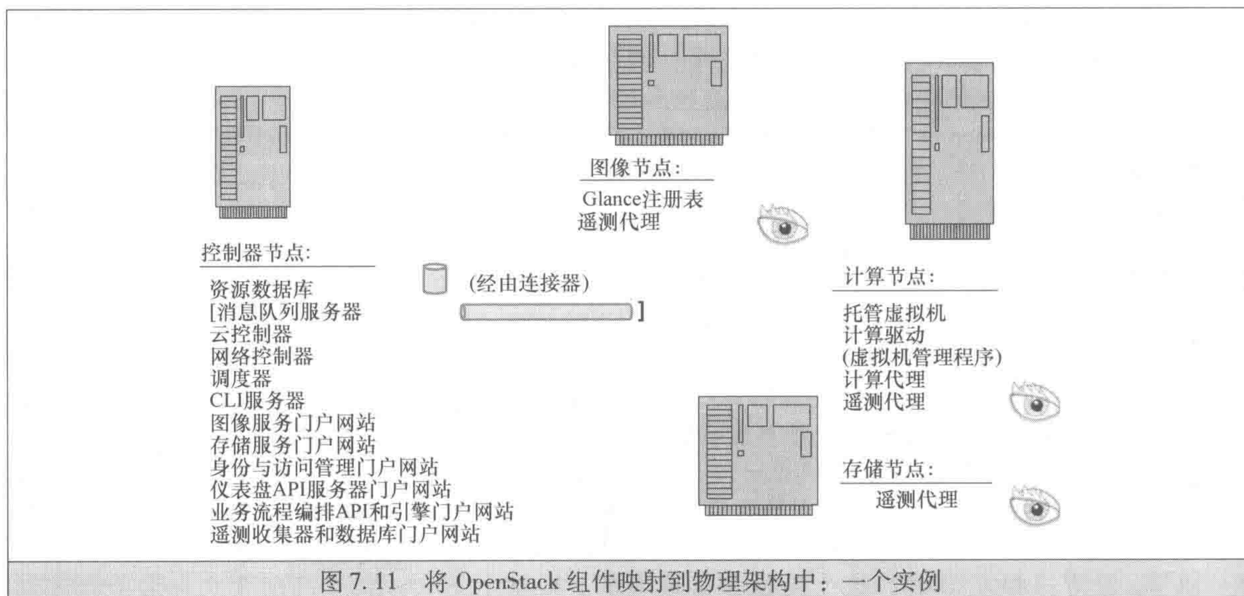


图 7.11　将 OpenStack 组件映射到物理架构中：一个实例

除了消息队列服务器（在括号中进行标注）外，它还涉及图 7.11 中的实体。我们暂时推迟讨论这一问题（在相当简单的部署实例中，所有控制器组件都在相同主机上运行），因为它似乎是多余的。在回顾组件之间的实际交互方式之前，我们将逐步介绍一种简单的事件流，它最终导致了虚拟机的生成：

1）事件流开始于远程用户使用提供虚拟机的请求，来调用控制器中的 API（更确切地说，是 Nova API 服务器）。然后，控制器请求调度器查询资源数据库，以确定合适的计算节点（在这种情形中，由于只有一个计算节点，因而这是非常容易的），且命令其计算代理来配置虚拟机㊀。

2）计算代理符合请求，查询资源数据库以获取有关图像的精确信息，从图像节点注册表中获取图像标识符，最后从存储节点加载图像本身，并命令虚拟机管理程序创建一个新的租户虚拟机。

3）计算代理将有关新虚拟机的信息传回到控制器节点，并请求网络控制器提供连接信息。

4）网络控制器更新资源数据库，并完成网络配置。

5）类似地，计算代理与卷控制器进行交互，以创建存储卷并将其连接到虚拟机。

毋庸置疑，为了简化讨论并完成基本流程，我们省略了几项基本功能。当然，这一简单序列不涉及业务流程编排，在这种情况下也是不必要的，这是因为：①“堆栈”只包含一台虚拟机而不是几台机器的基础设施；②假定由堆栈提供的服务不需要自动扩展或任何其他需要监视和自动干预的服务。当讨论一个更加复杂的实例时，业务流程编排将会很快进入我们的视线。

但最明显的疏忽是所有身份和管理相关事宜，包括对原始请求的认证及其授权。本书将在下一节中单独讨论这个问题，将会看到与身份管理相关的活动贯穿于所有步骤。

现在，在每个 OpenStack 组件内各个部分之间，准备进一步澄清应用级的通信特征。基本思路是任何两个对等体（即客户端和服务器）之间不存在共享数据㊁。

编写 OpenStack 软件的目的是创建高可用性系统。高可用性是可靠性大概念的一个方面，本章参考文献［5］将其定义为“系统恢复正确操作的能力，它支持系统在某些组件出现故障时恢复提供服务”。高可用性通过冗余和复制系统组成部分（在我们的实例中，是指网络、存储和计算组件）来实现，这些组成部分可能出现单点故障（Single Point of Failure，SPoF）。

复制服务器在称为群集的一组计算机上运行。所有这些服务器必须以客户端的一个服务器出现，如图 7.12 所示。因此，将其中的一台机器指定为代理服务器，负责在其他服务器之间分配客户端请求并均衡其负载。代理本身不是单点故障，因为集群中的其他每台计算机都要准备好承担前端和负载均衡功能。

㊀ 细心的读者可能会问这里使用的是什么协议。我们有意推迟命名这一协议和所有其他应用协议，直到完成对事件流的描述。

㊁ 需要注意的是，这不同于复制。例如，集群中的数据库服务器可能会共享其中的复制数据。



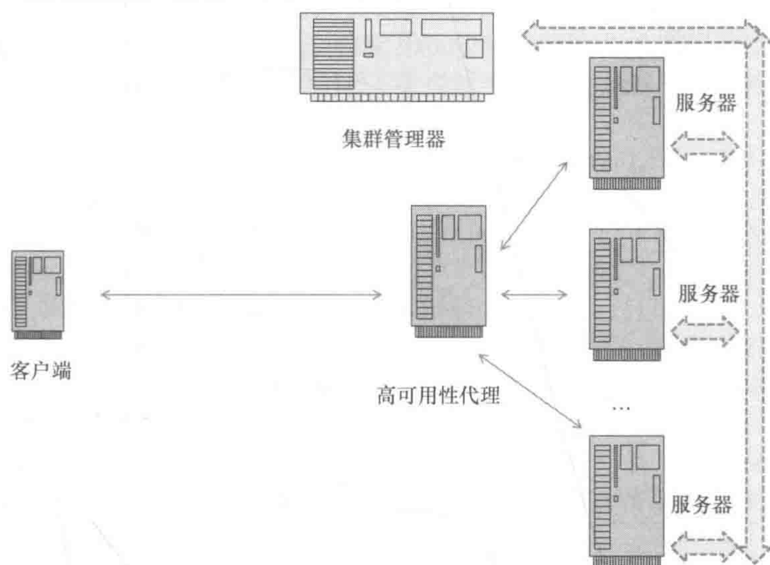


图 7.12 高可用性集群

顺便提一下，刚刚描述的机制与所谓的双活模式（所有服务器同时运行）相关，但并非在集群中实现高可用性的唯一机制。另一种模式是主动/被动模式，其中额外安排一台服务器保持备用状态（作为热备用）但不在线，且在活动服务器发生故障或过载的情况下联机。OpenStack 同时支持两种模式。

但是，集群何时“知道”将代理功能分配给另一台服务器，或者何时将热备用转为在线？为此，还存在另一种功能——集群管理器。其作用是根据配置观察集群运行是否正常，并根据情况重新对其进行配置。为了实现这一监控目的，OpenStack 目前使用名为 Pacemaker 的软件，它是集群实验室（Cluster Labs）开发的产品。

迄今为止，大家应该清楚为什么在客户端和服务器之间不能共享任何状态。假设客户端发送两条请求。第一条请求由代理指向图 7.12 中的顶级服务器，第二条请求由代理指向底部服务器。如果在客户端和服务器之间共享该状态，且由第一台服务器对该状态进行修改，则第二台服务器将无法知道状态已经发生改变！

正如多次提到的那样，设计万维网（World Wide Web，WWW）协议——HTTP（全球网络协议）完全符合这一目标：HTTP 客户端和 HTTP 服务器之间没有共享状态。在数十年的万维网实践中，业界已经学会如何开发和部署高效的服务器和代理。因此，获取所有可用和正确的软件，并将其应用（重用所有可用软件）到新领域意义重大，它比提供 Web 服务更为普遍。

这里的计算模型相当简单。HTTP 服务器是用于监听特定端口的守护进程（或线程）。当它获得一条消息（一个 HTTP PDU）时，它会对该消息进行解析，并完成一次性行动，最初只限于获取一份文件并将其传回到客户端，但随后（当详细讨论这一问题时将会看到）变得越来越复杂[㊀]。两台不同服务器可能在相同主机或不同主机上运行——服务接口保持不变。在专用主机上将全局管理服务组合起来是非常合理的，就像我们的部署实例中给出的情形一样。

为此，所有名称中包含 API（即 nova - api）的 OpenStack 模块都是提供 REST 服务的守护进程（将在附录 A 中进行讨论）。

通过高级消息队列协议（Advanced Message Queuing Protocol，AMQP）来执行守护进程之间的通信[㊁]。图 7.11 中的消息队列是支持此项功能的结构。

AMQP 可以从管道的任一端开始启动。相反，HTTP 事务只能从客户端启动，因为 HTTP 是纯客户端/服务器协议。

㊀ 是行动而不是协议！

㊁ 例如，前面提到的指向调度器的请求（事件流的第 1 步）通过采用 AMQP 来实现。nova - compute 守护进程通过虚拟机管理程序的 API 来创建和终止虚拟机实例，并从消息队列中获取一条请求，一次获取一条。

OpenStack 中的业务流程编排存在 3 个问题：第 1 个问题涉及云应用（即堆栈）的生命周期管理规范及其现实示例；第 2 个问题与堆栈状态监视有关，以符合运行虚拟机的规范；第 3 个问题与采取补救行动有关。正如前面提到的，Heat 组件负责解决第 1 和第 3 个方面，而 Ceilometer 则负责解决第 2 个问题。

Heat 的早期目标是与 AWS CloudFormation 兼容，以便能够将已与 AWS CloudFormation 协同工作的服务移植到 OpenStack 上。相反，可以将采用 OpenStack 在云中开发和运行的服务移植到 AWS 上。实际上，这种云可能会突然进入 AWS。为此，OpenStack 既认可 AWS CloudFormation 模板，又提供与 AWS CloudFormation 兼容的 API。

随着时间的推移，Heat 项目已经开发出自己的模板——准确地说，是 HOT，即 Heat Orchestration Template 的缩写。模板本身具有与 AWS CloudFormation 模板相同的格式和语义，但它是用一种称为 YAML 语言[⊖]而不是用 JSON 语言来描述的。不管怎样，模板是一种指定基础架构资源及其关系的文本文件。后一项功能实际上是程序性的，因为它可以决定创建虚拟机以及分配存储卷和网络连接的顺序。

另一项编程性特征可概括为模板具有动态特性：当模板发生变化时，Heat 需要相应地修改服务。将首先开始编排业务架构 Heat 部分的计算架构的描述（即计算进程方面的描述），如图 7.13 所示。

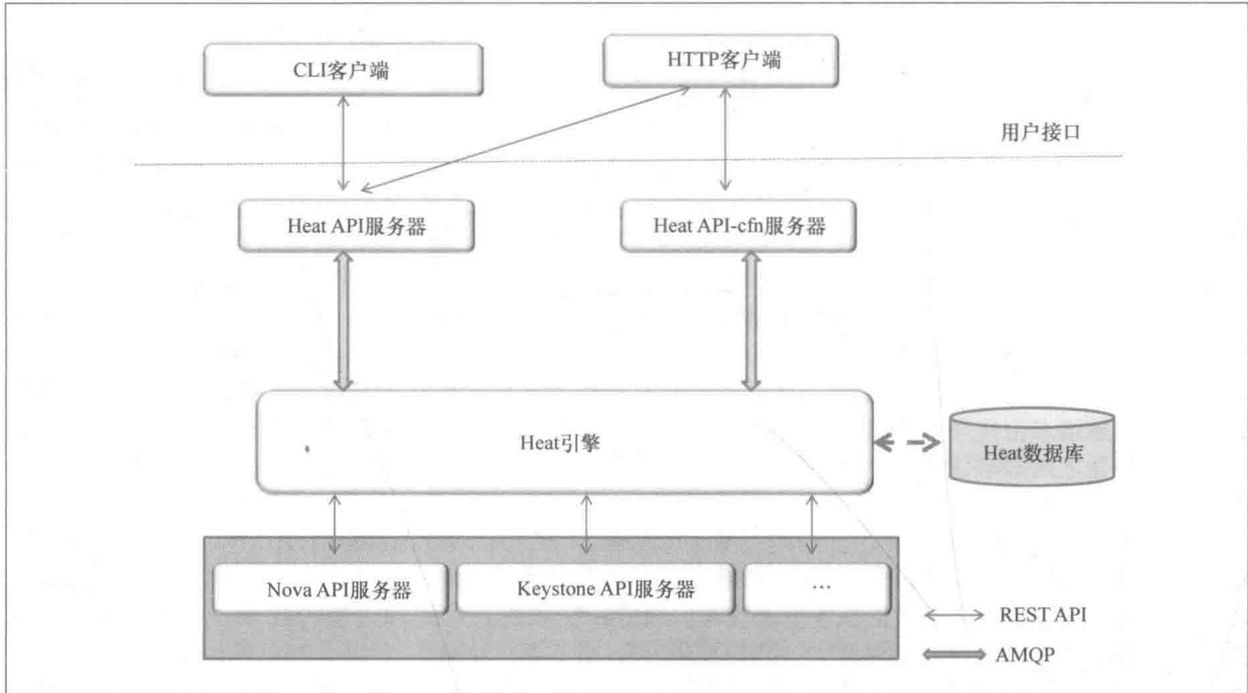


图 7.13　Heat 计算架构

Heat 引擎是负责根据模板规范启动堆栈的过程。用户接口功能由两台服务器（Heat API 和 Heat API - cfn）执行，它们分别为 HOT 和 AWS CloudFormation 兼容服务提供 REST API。这两者中的任何一种服务仅充当 Heat 引擎的前端，通过它和 AMQP 可以实现通信。

毋庸讳言，这种架构支持在可用机器之间以比早期控制器架构图提出的方式更加灵活的方式分配进程。所有相同类型的服务器都可以通过高可用性代理进行复制和访问。

除了 REST API 之外，Heat 组件还提供了命令行界面（CLI），但是 CLI 代理将命令转换为 REST API 业务流程编排，因而 Heat 引擎不直接处理 CLI。这些命令在语义上与 API 相同，因为它们指的是同一性能。接下来按照与模板、堆栈、资源以及与资源有关的事件的相关度的顺序来简要介绍命令：

1）template - show：为给定堆栈请求模板。

2）template - validate：请求模板使用特定参数进行验证。

3）stack - create：请求创建堆栈。

⊖ YAML 是左递归的首字母缩略词：它代表“YAML 不是标识语言”。

4）stack - delete：请求删除堆栈。

5）stack - update：请求更新堆栈（根据文件或 URL 中描述的数据或使用特定参数的新值）。

6）action - suspend 和 action - resume：分别请求活动堆栈执行中的同名操作。

7）stack - list：请求所有用户堆栈的列表。

8）stack - show：请求给定堆栈的描述。

9）resource - list：请求属于堆栈的资源集合。

10）resource - metadata：请求资源的元数据属性[㊀]。

11）resource - show：请求给定资源的描述。

12）event - list：请求当前堆栈中所选资源的事件列表。

13）event - show：请求特定事件的描述。

Heat 还提供可编程扩展的工具——资源插件，它扩展了基本资源类，并为上述命令实现了恰当的处理程序方法。

迄今为止，已经解决了所谓的 Northern API（用于业务流程编排的用户接口）问题。两台 API 服务器都充当 Heat 引擎的前端。为了执行这些命令，Heat 引擎依次调用 Southern API（Nova API、Keystone API 等）。简而言之，调用次序是由创建图像的早期工作流提出的。

OpenStack 业务流程编排工作的其余部分是用于支持告警的机制。事实上，服务模板的早期实例是指在 Heat 早期版本中部分模仿的 AWS CloudWatch 服务。用户可以使用创建和更新告警以及响应告警的机制。有关详细信息，以及对简单但深刻的用例测试的描述，强烈推荐 CERN（欧洲粒子物理研究所）达维德·米其里诺（Davide Michelino）的暑期学生报告。

然而，OpenStack 后来决定不再使用 AWS CloudWatch API，而是依赖于 Ceilometer。因此，这是在撰写本书时 OpenStack 所采取的行动。

现在讨论 Ceilometer。正如前面已经提到的，其目标是计量，即衡量资源的使用率。在电信业务中，计量是整个收费过程的第一步，而其他步骤是评级和计费。为此，Ceilometer 提供了评级引擎可用于开发计费系统的 API，但我们不会在此担心此事。我们关注的焦点（严格按照本书的目标）是使用计量来确定何时需要自动扩展操作。

因过去每个 OSS 项目存在的障碍而导致 Ceilometer 的任务复杂化：大型系统的不同部分总是使用不同方式来提供管理数据，且一些部分根本不提供任何数据。为了处理这一问题，OpenStack 必须创建若干种机制。在这些机制中，如果采用相同的方法，则一种机制即可满足要求。

Ceilometer 模型采用 3 种类型的参与方：各种遥测代理、遥测收集器和发布方。收集器汇集来自代理（这些代理负责管理 5 类 OpenStack 组件：计算、组网、块存储、对象存储和图像）的数据，然后将这些数据传输给发布方，发布方将其存储在数据库中或将其传输到外部系统。引入这一术语后，我们注意到，为了编写本书，收集器和发布方实际上是一个实体，称之为收集器。在图 7.11 相当简单的配置中，将代理描绘为眼睛，并假定以通知的形式将其数据发送给收集器。

在理想系统中，每种组件都有一个代理，用于通过统一消息系统（称为 Oslo 总线）发布事件通知。有关详细信息，请参阅相应的 OpenStack 文档，但需要注意的是，AMQP 与包含其他消息传递机制的 Oslo 总线兼容。为了简单起见，假设所有消息都通过 AMQP 进行发送。

第一种（优选）机制受总线侦听器代理影响，它负责处理所有通知事件并产生 Ceilometer 样本。同时，在所有组件都能发布事件通知的系统中，这是唯一必需的机制。

没有统一的通知实施方案，第二种（次优选）附加机制是有序的。这里，实际用于创建通知的推代理需要添加到每个受监视的节点上。且如果由于某种原因而导致无法实现，则就会转向第 3 种机制——轮询代理。

轮询代理只是完成其名称所暗示的动作：它检查环中的远程节点（通过 REST API 或 RPC），其中包括等待指定的间隔[㊁]。这是最不推荐的方法，因为其实现对弹性具有负面影响。

㊀ 此概念和功能来自 AWS CloudFormation 模板，它支持添加资源的描述类元数据属性，该属性将资源与结构化数据关联起来。

㊁ 同时，靠近硬件端的是检查设备，它们使用各种协议（如 SNMP）来采集数据，然后将其传输给代理。

代理的功能和运行位置有所不同。例如，计算代理在计算主机上运行。中央代理是中央管理系统的一部分（根据之前的命名，应当将其称为控制器节点）。其职责包括控制节点或实例以外的资源。当然，收集器也在中央管理系统上运行。应该说明的是，可能存在着若干台运行在中央管理系统不同部分上的不同主机（或者在高可用性集群中，每台主机都可以运行所有这些部分）。Ceilometer 还支持对代理进行配置。

将收集的数据存储在 Ceilometer 数据库中。Ceilometer 提供两组 API：①用于写入收集器的 API；②用于访问数据库的 API。与所有其他组件一样，API 服务器是一种独立进程。

图 7.14 对上述架构进行了归纳总结，它描述了一种逻辑消息流。实际上，所有组件都通过真实的消息总线进行“互连”，也就是说，它们读取和写入相同的逻辑“线”。

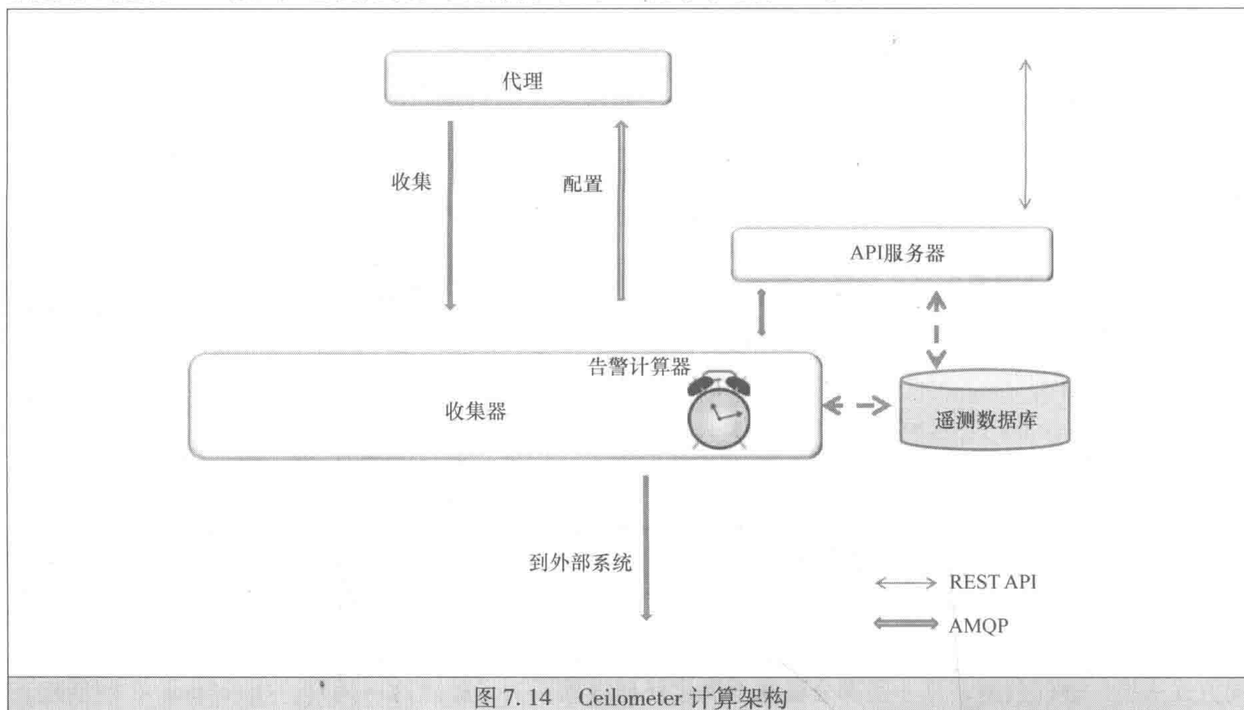


图 7.14　Ceilometer 计算架构

Ceilometer 的另一个特征（对我们来说，最重要的一项功能）是根据预定义阈值创建告警的能力（如“告诉我何时 CPU 利用率达到 70%”）。OpenStack 专门针对这一能力定义了一个单独模块[㊀]，但是为了简单起见，将告警计算器看作是收集器的一部分。

将使用图 7.15 来准确说明遥测与业务流程编排（即 Ceilometer 和 Heat）之间的互连关系。回到讨论 AWS CloudFormation 时引入的自动扩展实例。

为了知道何时放大（或缩小）堆栈，Heat 引擎需要关于堆栈实例 CPU 利用率的反馈信息。为了实现这一点，可以根据计算代理的指标来定义告警。在 Ceilometer 的支持下，可以根据模板指定的规则对指标进行评估，并将异常情况报告为告警通知。当然，Ceilometer 本身并不负责读取模板——只有 Heat 引擎才能读取模板。Ceilometer 只需为 Heat 引擎提供一种足够灵活的 API，来表达自动扩展组的模板规则。

在左上角，对 HOT 模板的一部分进行定义，并基于自动扩展组的 CPU 利用率来设置告警。也就是说，当 CPU 利用率超过 70% 时，服务器组“扩大”，即添加新的服务器实例。OpenStack 手册将特定属性——alarm_actions 定义为“当状态转换为告警时调用的 URL（webhook）列表（这是它的原意!）”。模板还指定了收集指标和所使用统计信息的周期（例如平均值）。

当堆栈创建完毕后，即可根据 Heat 引擎的请求来设置告警。当告警“发声”时，Heat 引擎将通过创建新实例来采取进一步行动，将其连接到网络等。类似地，可以将告警设置为按比例减小（即当平均 CPU 利用率低于 40% 时），因而当额外实例变得冗余时，即可将其取消。需要注意的是，此项功能不仅表现在云提供商的运行效率，而且它还为客户节省了资金，客户通常会为每一实例向云提供商支付费用。这是一个真正的弹性实例。

㊀ Ceilometer 告警器服务。



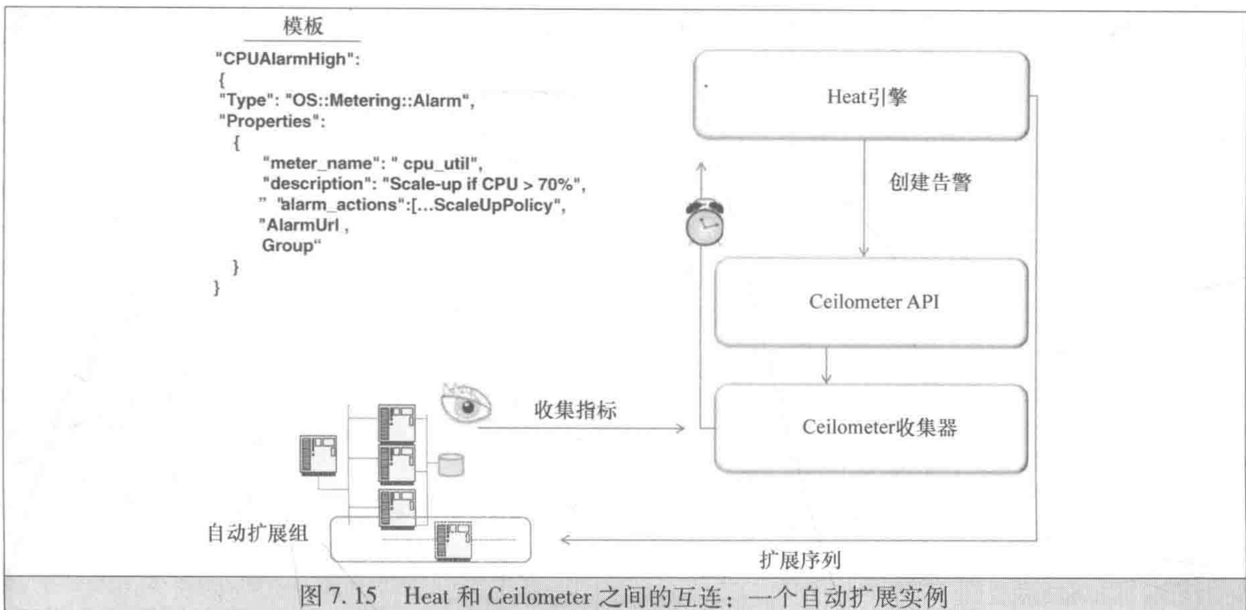


图 7.15 Heat 和 Ceilometer 之间的互连：一个自动扩展实例

重要的是，需要注意 OpenStack 支持软件配置和管理工具的集成——特别是 Chef 和 Puppet，在上一节中曾经提到过这一点。

前面讨论了工作流工具。在编写本书时，OpenStack 正在积极地开发一种工作流工具，并将其作为 Convection 项目的一部分。在项目说明中，工作流被称为任务流，且计划是提供任务流即服务（Task Flow as a Service，TFaS），这与 Amazon AWS 的做法类似。

服务的愿景是让用户能够写入和注册工作流。然后，应用可以调用此工作流（且稍后还会检查其状态或终止工作流）。业务流程管理器的工作是对工作流中的每次状态变化做出反应，并调用相应任务。

该服务是递归的，因为 Heat 本身可以使用 TFaS 来管理自己的任务。例如，任务流可以调用 Heat API 来启动给定任务。根据这一愿景，“业务流程编排涉及智能创建、组织、连接和协调基于云的资源，这可能涉及创建任务流和/或执行任务。”

通过结合并展示迄今为止描述过的组成部分之间的相互作用，图 7.16 总结了作者对 OpenStack 业务流程编排愿景的理解。同时，在编写本书时，这只是未来发展的蓝图。

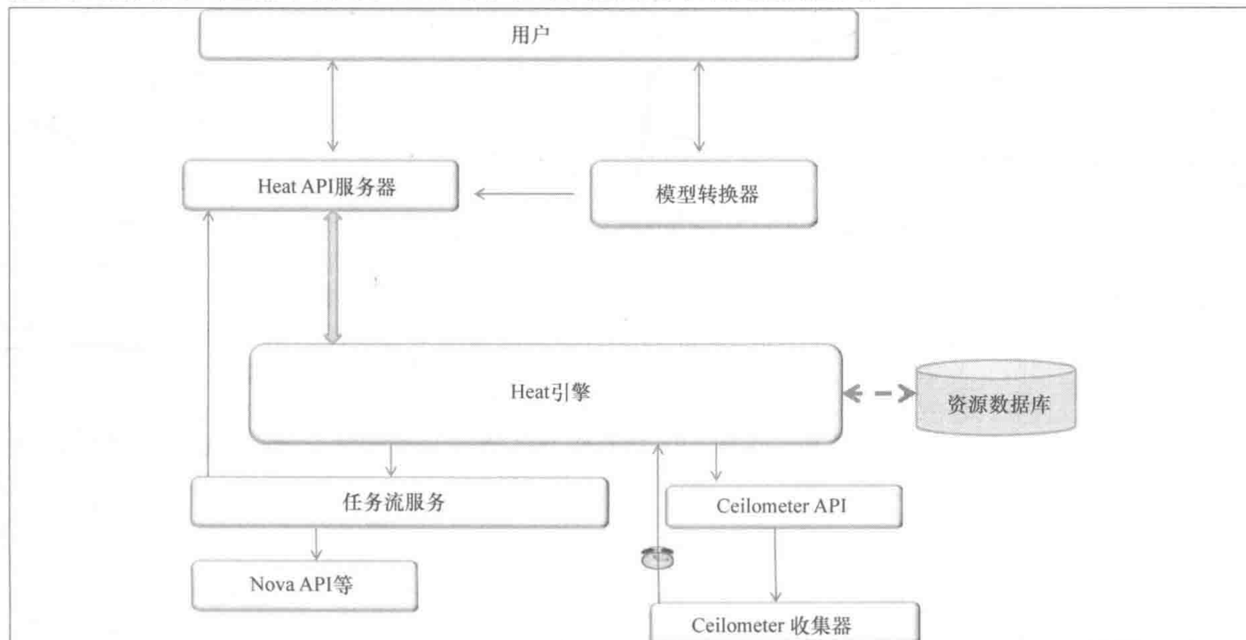


图 7.16 集成业务流程编排架构

右上角描绘的部分是其他模板规范（cfn 或 TOSCA）的泛称，每种规范都由一种被称为模型转换器的实体来解释。本书将在附录 A 中回顾一些规范实例。

通过考虑另一个部署实例（从组网的角度）来结束本节内容，如图 7.17 所示。该图对图 7.11 进行了修改，来说明云数据中心内现代节点的结构。就像在人体解剖学书中一样，图 7.17 描述了身体的不同层次（如肌肉或骨骼）。这里来观察一下互连组件的实用方法。为减少杂乱，省略了图像节点，但是所有其他节点都存在——现在被复制并假定在集群中工作。

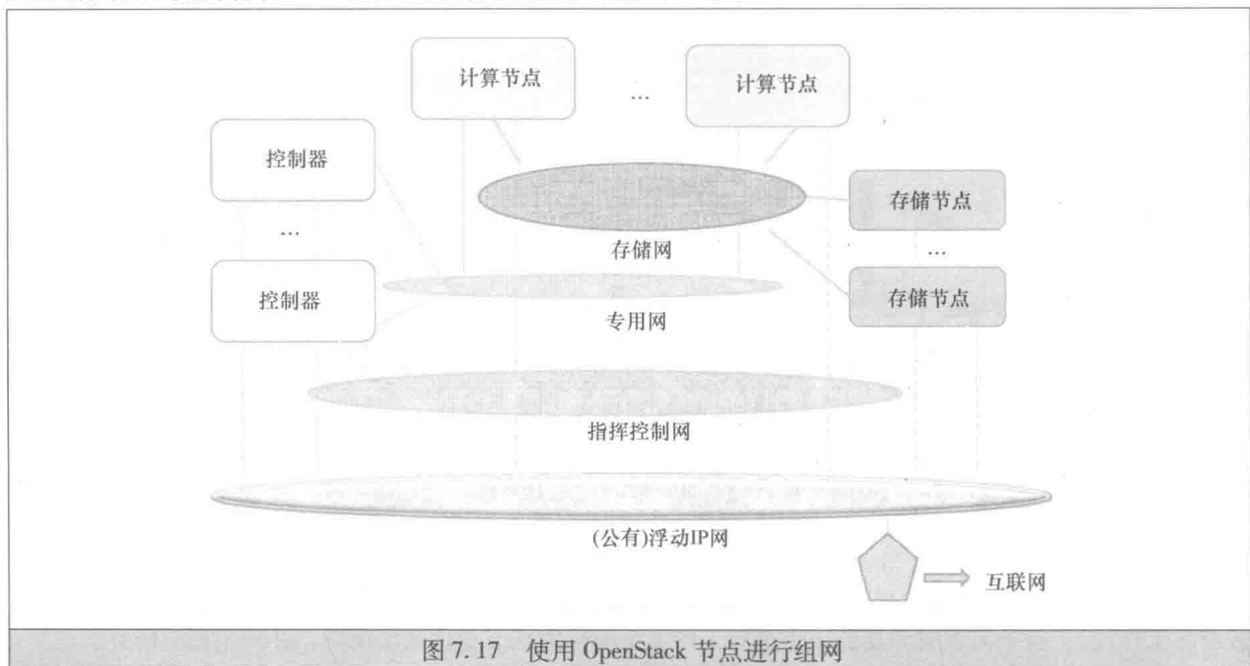


图 7.17　使用 OpenStack 节点进行组网

以下 4 种（第 2 级）网络相互之间是完全独立的：

1）存储网仅用于访问存储，因而仅用于计算节点和存储节点的互连。

2）专用网仅用于托管虚拟机之间的通信。

3）指挥控制网络仅用于业务流程编排和管理。

4）公有网允许连接到互联网。正是由于这个原因，公有网使用如前所述的浮动 IP 地址。

将这些网络分离开来或多或少具有一定的典型性[⊖]——无论是出于安全方面的考虑还是用于区分容量，因为不同用途拥有不同的带宽需求。作为最低限度，专用网、公有网和指挥控制网至少是可取的。

7.4　认证与访问管理

身份和访问管理（Identity and Access Management，IAM）学科主要负责处理认证和授权事宜。两者都是云计算不可或缺的：在虚拟机（或堆栈）的整个生命周期内，对虚拟机（或堆栈）应用任何操作之前，管理和业务流程系统必须知道谁正在请求操作以及请求实体是否拥有操作权。

对身份和访问管理（IAM）的需求是泛在的，而且在其他情境下是显而易见的。每台机器（无论是物理的还是虚拟的）均需要拥有自己的 IAM 机制（操作系统的一部分），来控制对程序和数据的访问。应用还可以拥有自己的 IAM 来控制对其服务的访问。不同情境中的 IAM 功能在权限方面是分级的；与云管理系统相关的是享有最高特权的。这与虚拟主机的管理权限层次结构类似，如图 7.18 所示。稍后会进一步解释采用管理权限层次结构的原因。此处暂时回顾一下第 3 章的内容，来介绍特权级别和保护环的概念。

⊖ 推荐学习 Mirantis OpenStack®的参考架构，文档的第 17 页给出了一种包含 5 种网络的架构——在该实例中，行政和管理网络是分开的。

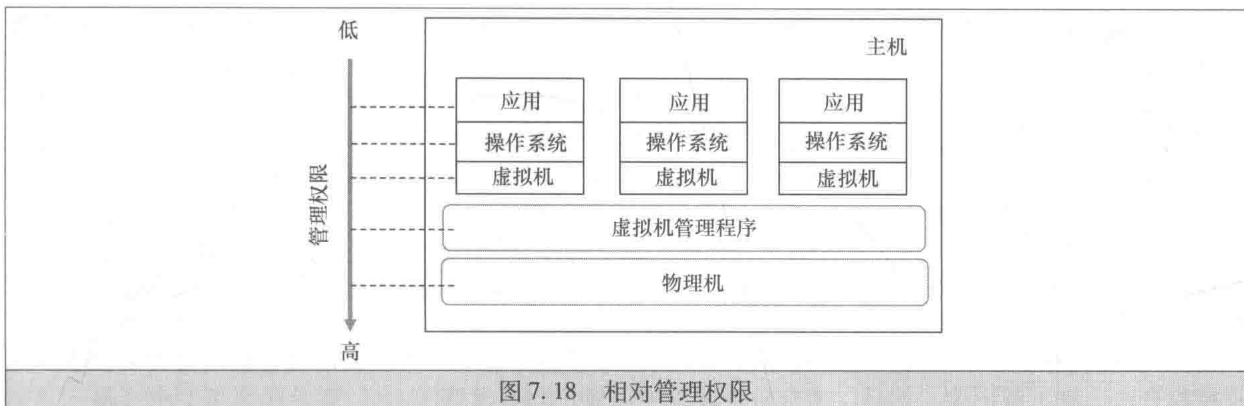


图 7.18　相对管理权限

图 7.19 归纳了 IAM 领域包含的主要内容。简而言之，它涉及身份信息的生命周期和相关性，以及实体的认证和授权。这些身份信息代表与实体对应的不同角色。将实体可以被全面一致识别为唯一的构造称为身份。身份可能与个人、项目、过程、设备或数据对象相关联。

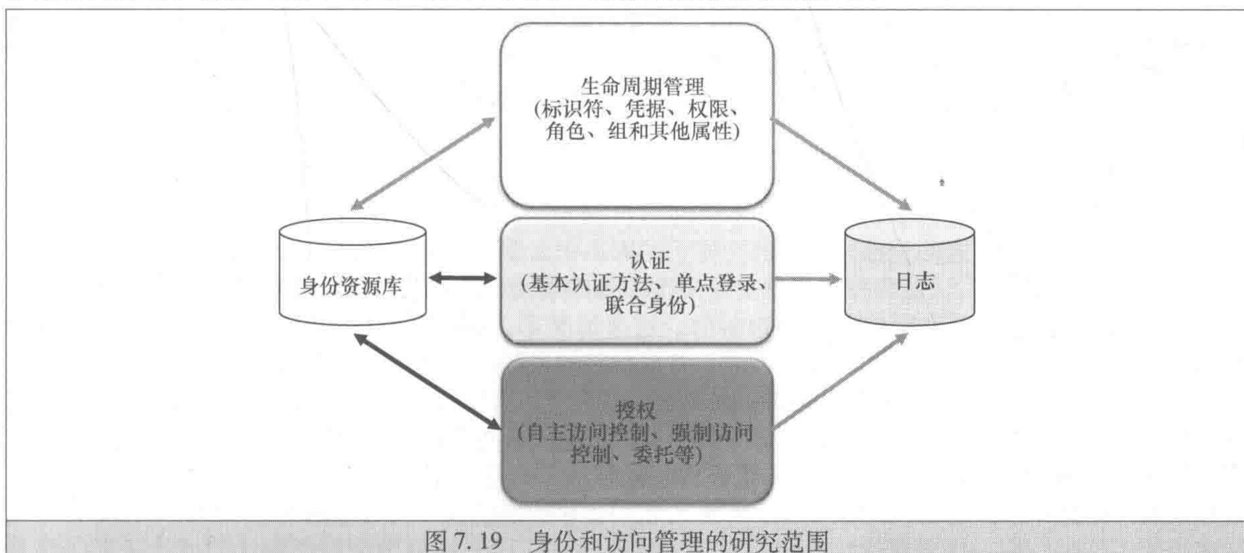


图 7.19　身份和访问管理的研究范围

将身份定义为一种用于认证实体的实体名称（标识符）[㊀]和凭据结合起来的结构。为给定实体创建身份需要凭据证明、角色设置和提供资源库中的相关数据（包括权限）。自然地，必须有足够合适的安全和隐私控制来保护所有生成的身份相关信息。为此，有必要为系统之间如何使用、存储和传播这些信息制定明确的策略。这种策略既可以与提供商有关，又可以由政府规章（如美国的“萨班斯－奥克斯利（Sarbanes－Oxley）法案”）明确。为支持审计和报告的策略合规性，必须记录关键的 IAM 活动（以在线和离线的形式）。该系统使用人员的审计线索以及做出何种授权决定对于事件管理和取证也是至关重要的。

在本节中，将首先讨论云计算与身份和访问管理有关的含义。然后，将通过描述其构建块来讨论最相关、最先进的 IAM 技术。这里只提供框架，并将在附录 A 中详述细节。最后，作为个案，将研究 Keystone，一种用于实现身份和访问管理的 OpenStack 组件。

7.4.1　云计算的含义 ★★★

为了帮助理解云计算的含义，考虑一种通过基于 Web 的门户网站来创建虚拟机的流程。假定爱丽丝是门户网站的用户[㊁]。创建流程如下：

㊀ 实体可能有多个名称（考虑到别名）。在这种情况下，可以链接多个标识符。

㊁ 这不仅解决了选择性别名词的问题，而且在安全文本中也是众所周知的惯例，将参与方称为鲍勃和爱丽丝。鲍勃将在下一章出现。

首先，爱丽丝试图访问该门户网站，而该门户网站启动了认证和授权步骤。爱丽丝向门户网站提供其凭据。

此时，门户网站会认证爱丽丝身份并确定其权限。但是，作为用户接口，它依赖于另一种云组件（即身份控制器）来实现。门户网站通过身份 API 来调用身份控制器。

认证和授权成功后，门户网站将为爱丽丝提供她应当拥有的服务和资源。接下来，依托门户网站，爱丽丝要求为虚拟机配置某些属性。这将触发爱丽丝在各种组件上的一连串动作：

1）门户网站通过计算 API 构建请求，并将其发送给计算控制器。该请求中包括爱丽丝的凭据。

2）计算控制器对凭据进行验证[1]。如果一切顺利，则它将分配 1 个计算节点，并命令该计算代理配置虚拟机。

3）在接到命令后，计算代理通过图像 API 向图像存储器请求图像。该请求包括爱丽丝的授权。如果授权合法，则下载图像。然后，虚拟机管理程序创建一台新虚拟机。

4）虚拟机创建成功后，计算代理（通过组网 API）请求网络控制器提供指定连接。该请求包括以一种网络控制器能检查的形式呈现的对爱丽丝授权的确认信息。

5）成功完成授权验证后，网络控制器分配网络资源，并将相关信息返回到计算代理。

6）计算代理请求卷控制器创建所需的卷，并通过块存储 API 将其连接到虚拟机。

7）卷控制器分配卷并将相关信息返回到计算代理。（当然，这一步也需要授权检查）

现在，可以将虚拟机已经配置完毕以及用于访问它的相关信息（如 IP 地址和根密码）通知给爱丽丝。

从上述流程[2]可以得出如下 5 点结论：

1）只有那些通过身份验证的用户可以访问门户网站。因此，爱丽丝首次尝试访问时，她必须登录，提供用户接口所规定的凭据。所需的具体凭据取决于云服务支持的认证方法。

2）爱丽丝登录后，认证和授权步骤重复多次。具体来说，当涉及 API 请求时，爱丽丝的特权验证是有序的。需要重复认证和授权是可以理解的。通常情况下，认证和授权的影响仅限于 API 事务和分布式系统中的相关组件。需求要进一步考虑不共享状态的目标，以支持“大规模可扩展性”。

3）爱丽丝不必在每次需要凭证时都提供其凭证。这是一个重要的设计问题，可以通过实施单点登录来解决。因此，在爱丽丝登录之后，需要其凭据的其他交互可以在爱丽丝不干预的情况下进行处理。

4）需要一种授权机制来支持云服务组件代表爱丽丝采取行动。显然，将爱丽丝的原始凭据复制到分布式组件中是不可接受的（想象一下，将你的密码提供给您委托任何任务的每个人）。一种可行的机制应当使用临时结构来代替爱丽丝的原始凭据。毋庸讳言，这一结构至少与爱丽丝的原始凭据一样具有良好的安全性，这一点是至关重要的。不过，奇怪的是，这一要求现在看来可能有些奇怪，稍后将证明（见附录 A）这一要求可以得到满足。

5）除了爱丽丝，还存在其他特权用户，如云管理员或进程所有者。他们也需要进行认证和授权。

一旦自动化成为云计算的标志，事情就会变得复杂。例如，考虑自动扩展，就会得出这一结论。正如所看到的，业务流程管理器负责创建新虚拟机以响应某次告警。当现有虚拟机的负载达到预设阈值时，通常就会触发此类告警。创建虚拟机的流程与直接涉及爱丽丝的流程几乎相同。这里的主要区别在于，是业务流程管理器而不是门户网站通过计算 API 来构建和发送请求到计算控制器。业务流程管理器代表爱丽丝采取行动。它从事先提供的模板处获取虚拟机的规格。如前所述，为了做必要的事情，业务流程管理器需要在计算控制器的请求中包含爱丽丝的授权，且业务流程管理器应该使用一种用于代替爱丽丝原始凭据的结构，该结构应当在短时间内有效。

但这里存在一个问题。规模扩展告警的时机是不可预知的。很可能当告警出现时，现有临时结构已经过期。因此，业务流程管理器需要动态（按需）获取新的临时结构。

此外，自动扩展不仅仅是配置和启动新虚拟机。启动虚拟机的原因是运行应用。一旦虚拟机启动并运行，业务流程管理器还需要远程安装应用、提供与应用相关的数据（如应用用户和管理员的凭据）、配置系统和应用程序，并最终启动应用。为了执行这些任务，业务流程管理器需要访问云中预先

---

[1] 认证和授权可以由计算控制器之外的组件（如身份控制器）来完成。

[2] 就身份和访问管理而言，它基本上拥有与其他操作相同的流程，如启动、暂停、停止和取消配置。

设置的与应用有关的数据（即元数据）。然后，需要在虚拟机上运行脚本，这可能需要特殊权限。幸运的是，假设爱丽丝获得了适当权限，则业务流程管理器可以完成所有操作。

但这又会出现另一个问题。首先，业务流程管理器必须拥有对爱丽丝虚拟机的特权。因此，爱丽丝必须能够以足够的粒度委派角色来降低潜在安全风险。此外，由于业务流程管理器假设特殊用户对虚拟机拥有任何权力，因而分配给特殊用户的权限应当是执行手头任务所需的最低权限。

总之，云管理员可能会滥用权限。因此，在云环境中，必须制定可以限制潜在损害的控制措施。

7.4.2 认证 ★★★

当某人需要向计算机系统进行身份认证时，存在3种常见的凭据类型，如图7.20所示。爱丽丝可以基于她所知道的信息（如密码）、她所拥有的东西（如硬件安全令牌）或她自身的特征（如她的指纹）进行认证。显然，基于爱丽丝所知道的信息进行认证是最简单的方法，因为它不需要额外的小工具。这就解释了为什么密码应用最为广泛。

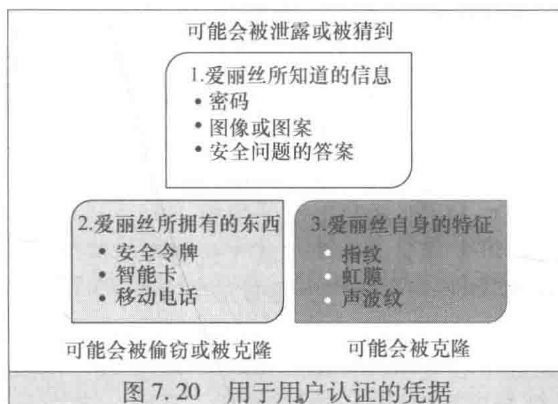


图7.20 用于用户认证的凭据

认证方法的选择取决于可以容忍的风险。显而易见，用户权限越高，认证方法应当越健壮。想象一下，攻击方伪装成云管理员可能造成的破坏有多大！

基于多种类型凭据（称为多因素认证）的认证可以显著提高保证水平。鉴于所涉及的各种特权用户，云基础设施服务至少支持双因素身份验证是非常有意义的。

双因素认证最常见的形式是将密码（即爱丽丝所知道的信息）和从设备得到的动态生成的代码（即爱丽丝所拥有的东西）结合起来。

当网络存在时，有必要使用加密系统。任何这种系统要么使用共享密钥（对称加密），要么依赖于公钥密码体制。

公钥密码体制需要一对密钥：一种被称为公钥，可以将其共享；另一种被称为私钥，必须进行秘密保存。假定公钥正确生成，则从公钥导出对应私钥在计算上是不可行的。一种得到广泛使用的知名公钥密码系统是RSA，它是由李维斯特（Rivest）、沙米尔（Shamir）和阿德曼（Adleman）于1977年在麻省理工学院开发的（因而算法由3个人名缩写命名）[23]。从公钥导出私钥在计算上不可行的属性是基于大数分解的难解程度。RSA的重要特征之一是公钥和私钥计算是可交换的：使用私钥加密的量可以用公钥进行解密。这一特征支持我们不仅可以使用该方案进行加密，而且还可以使用该方案进行签名和认证。

图7.21描述了公钥密码体制在认证中的应用。如图7.21所示，爱丽丝宣布自己的身份。在收到爱丽丝的消息后，认证系统向她发送挑战 R，它通常是随机数和时间戳的组合。爱丽丝使用 PrK_A（R）来响应，这是她使用私钥 PrK_A 对挑战进行加密的结果。然后，系统使用爱丽丝的公钥来对 PrK_A（R）进行解密。如果解密的量与原始挑战相同，则认证成功。不用说，需要假设爱丽丝的公钥可应用于计算机系统。如何实现是另外一件事，稍后将进行讨论。

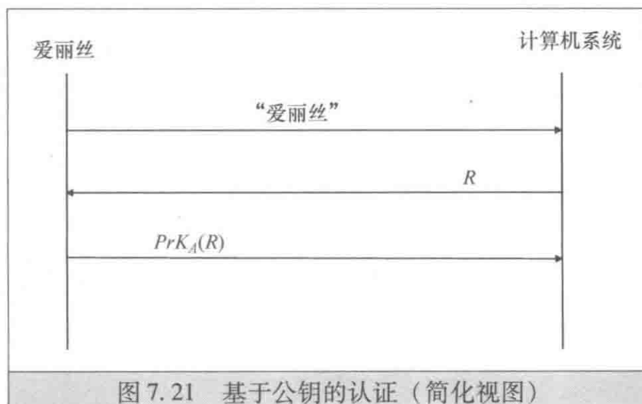


图7.21 基于公钥的认证（简化视图）

这里的一个重要细节是用于认证的密钥对必须仅限于特定目标，且不能用于数字签名。数字签名包含的操作与认证相同，即使用爱丽丝的私钥对文档或其摘要进行加密，然后使用爱丽丝的公钥对其进行解密和让其他任何人进行验证。如果使用同一密钥进行认证和数字签名，则可能会欺骗爱丽丝进

行数字签名，如将借据作为挑战骗爱丽丝签名，如图 7.21 所示。

不仅用于完成认证和数字签名任务的密钥对是不同的，而且密钥生成和管理过程也需要有所不同。虽然可以将用于认证和解密的私钥进行托管（即存放在可信的第三方处），但是某些管辖权限禁止将用于数字签名的私钥进行密钥托管㊀。在云计算中，密钥可以高效地被托管，因为拥有特权的云管理员可以访问所存储的私钥。在法规遵循方面，密钥托管是有问题的。为了满足这些要求，通常使用硬件安全模块——一种拥有管理接口的存储设备，该接口独立于虚拟机管理程序。

采用公钥密码体制，需要一方存储私钥，另一方要安全分发公钥（即证明爱丽丝及其公钥的关联性）。前一个问题稍微简单一些，因为它可以通过使用诸如智能卡等专门设备来解决。然而，自从威特菲尔德·迪菲（Whitfield Diffie）和马丁·爱德华·赫尔曼（Martin E. Hellman）[24]首次公开发布公钥密码体制以来，后一个问题始终是一大挑战。最终，行业公钥基础设施（Public Key Infrastructure，PKI）解决方案需要一种管理基础设施以支持：

1）发布将公钥与身份标识及一组属性绑定起来的 PKI 证书。

2）维护证书的数据库。

3）指定用于验证证书的机制。

4）指定撤销证书的机制（包括存储证书）。

在这种解决方案中，证书充当的是凭据的角色。劳伦·科恩菲尔德（Loren Kohnfelder）于 1978 年在麻省理工学院的学士论文[25]中介绍了该解决方案的一些核心要素，包括证书的结构、使用可信机构［或认证中心（Certification Authority，CA）］的数字签名来封装绑定的思路，以及证书撤销的概念。当然，证书应当规定一定的有效期。然而，这并不太好。虽然证书仍然有效，但是使绑定失效可能会出乱子。例如，攻击者可能得到私钥。证书撤销支持认证中心作废证书。

使用 CA 的签名将公钥与身份绑定起来会导致如何验证数字签名的问题。如果签名算法是基于公钥密码体制的，则该问题与公钥分发相同。因此，它可以通过相同的方法来解决：级别较高的认证中心向级别低的认证中心颁发数字证书。该步骤可以根据需要重复多次，从而产生一连串证书（或一个信任链）。图 7.22 从概念上给出了一个信任链。链的顶部必定是拥有根证书的信任锚点，该证书进行过自签名。假定任何人都可以生成密钥对和自签名证书，这些信任锚点必须是众所周知的，且数量很少。ISO 和 ITU - T 已经对层次认证模型进行了标准化，并发布了 ITU - T X. 509 建议书[26]。除其他内容外，X. 509 证书可以获取持有人的姓名和公钥、签发认证中心的姓名和签名、签名算法、有效期和签发认证中心的数字证书。尽管证书格式还有其他标准，可是 X. 509 应用最为广泛。它的成功是由于其内置的可扩展性。其他标准化组织（特别是 IETF）已经对扩展进行了详细规定。

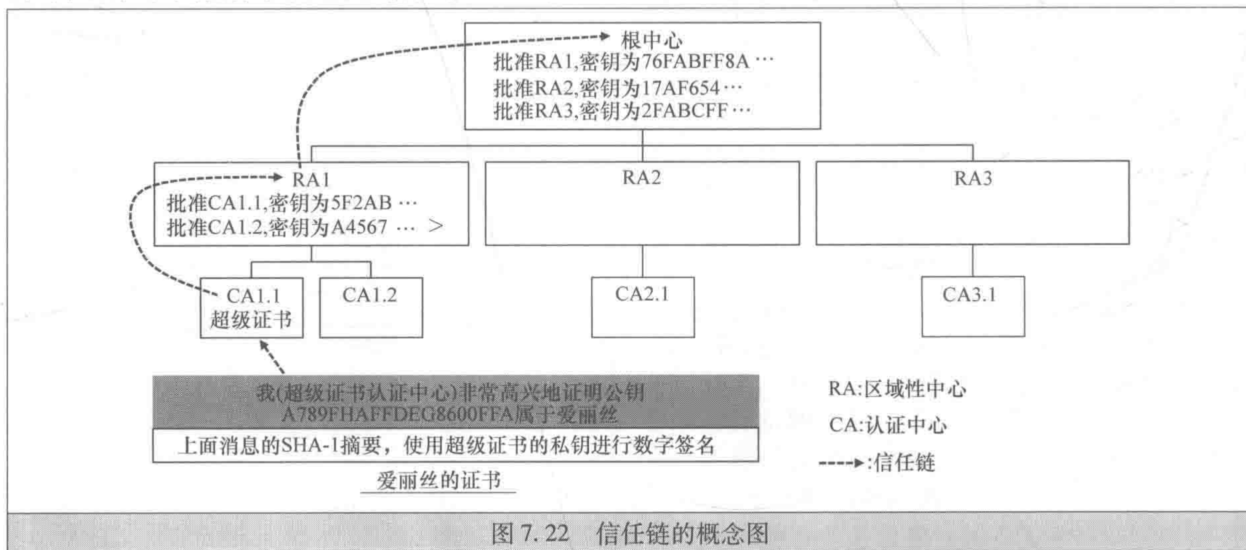


图 7.22 信任链的概念图

㊀ 例如，新泽西州曾经禁止托管用于数字签名的私钥；最近的新泽西州通知标题：181 加密和数字签名政策更加变本加厉，明确规定“用于数字签名、数字证书和用户认证的密钥不得包含在与第三方的密钥托管约定中”。

我们想强调的是，Web 安全性取决于这一标准：传输层安全（TLS）协议使用 X. 509 证书进行服务器认证[㊀]。自然地，X. 509 依赖关系转移到云计算。事实上，依赖性变得更强，因为证书不仅用于服务器认证，而且还用于客户端认证[㊁]。

在没有 PKI 的情况下，也可以使用公钥密码体制。安全外壳（SSH）协议就是一个例子，它在云计算中是不可或缺的。首先，SSH 是自动访问虚拟机的主要安全手段。根据 RFC 4252 和 RFC 4253 的规定，SSH 支持使用证书进行客户端和服务器认证。

安全外壳（SSH）标准存在的一个问题是它引入了多种选项。虽然它支持使用证书，但并不是强制性的，且导致相当不确定的部署场景。特别是服务器在收到连接请求时，可能仅发送公钥和密钥的散列值（也称为指纹）。客户端在首次接收时，可能会盲目接受公钥。这种首次使用信任（Trust On First Use，TOFU）的方法易于受到严重的攻击，但实践中仍在使用这种方法。

我们的观点是，在云计算中，这种做法必须改变。一种合乎逻辑的解决方案是在客户端和服务器认证中强制使用证书。遗憾的是，迄今为止这还是不现实的。

开源软件支持云计算，但是开源 SSH 软件（包括与所有 Linux 发行版捆绑的 OpenSSH）中实现的证书格式是专有的。

最佳做法是在安全数据存储 [如 RFC 4255 中定义的 DNS（Domain Name System，域名系统）] 中保留散列值的副本，这样客户端在与服务器的首次接触中可以对其进行查询。客户端验证从服务器接收的指纹，并将其与安全存储中的指纹进行比对。如果指纹匹配，则继续进行连接。

7.4.3　访问控制 

跟踪授权对象访问何种内容的方法之一是维护一张表（称其为访问控制矩阵），其中行对应于主体，列对应于数据对象。矩阵反映了系统的保护状态。主体是主动实体（如用户或进程），而对象是被动对象（如文件）。矩阵中的每项（用 A_{ij}来表示）规定了主体（用 S_i来表示）对给定对象（用 O_j来表示）的权限（如读取或写入访问）。表 7. 1 给出了这种矩阵，其中主体 2 权限最低，只有访问对象 1 和对象 2 的权限。相比之下，主体 1 权限较高，它拥有对所有对象的访问权限。事实上，主体 1 可能是系统管理员。由于最低权限原则，因而他只能读取对象 3（如工资单数据）和对象 4（如审计线索）。主体 3 可以对这些对象进行读写（该对象可能是一个进程）。矩阵为设计访问控制系统和确定系统是否安全提供了一种功能强大的模型[27]。

表 7. 1　访问控制矩阵

	对象 1	对象 2	对象 3	对象 4
主体 1	读取、写入、执行	读取、写入、执行	读取	读取
主体 2	读取、执行	读取、执行	无权限	无权限
主体 3	读取、执行	读取	读取、写入	写入

访问控制矩阵自然是稀疏的，因而需要考虑特殊的实现方案。简而言之，只需存储非空矩阵元素，但需要记录其位置。两种常见方法分别通过列和行来存储相关单元。列方法可以得到访问控制列表（Access Control List，ACL），行方法可以生成能力列表。本书将在附录 A 中对这些内容进行更为详细的讨论。

访问控制既可以是自由的，又可以是强制性的，这取决于谁拥有设置访问对象权限的能力。使用自主访问控制（Discretionary Access Control，DAC），对象所有者可以随意更改对该对象的访问权限。自主访问控制可以在大多数使用访问控制列表的操作系统中实现。在该方案中，对象所有者可以特别容易地部分或全部撤销赋予任何主体访问对象的权限。对象所有者只需从对象的访问控制列表中删除所关注的主体权限即可。然而，访问控制列表具有局限性。需要注意的是，自主访问控制不适合处理用户需要在一段时间内将某些权限委托给另一个用户的情况。毋庸讳言，动态授权（这在云计算环境中

㊀ 其工作机理与其前身——安全套接层（Secure Socket Layer，SSL）协议相同。

㊁ 读者可能会问，为什么基于数字证书的认证还没有替代人类用户传统的认证方法。一个主要原因是让人们安全保护其私钥非常困难。然而，私钥不受损害是 PKI 安全的基础。另一个主要原因是数字证书需要花钱。

是至关重要的）必须依靠一种独立机制来实现。

正常系统中的大量访问控制列表也使得确定每个用户的权限变得非常困难。当需要快速更新用户权限以反映人员或事件的突然变化时，这就容易出现问题。使用组结构是一种巧妙的解决方案。访问控制列表既可以包含用户，又可以包含用户所属的组。如果撤销用户的权限，但不撤销组的权限，则用户仍可以访问该对象。显然，该方案还有改进的余地。

在非自主访问控制中，即使对象是由主体创建的，主体可能也无法更改对该对象的访问权限。拥有此类权限的机构存在于系统层面。强制访问控制（Mandatory Access Control，MAC）是限制军事和政府机构中不同级别人员之间信息流的一个实例。通常，强制访问控制要求根据某一策略，系统中认证中心需要为每个主体（以及计算机系统中的每个对象）分配层次结构中的某个安全级别。访问资格是基于主体与对象所分配安全级别之间的优势关系来确定的。当且仅当对象在层次结构中级别相同或较低时，主体才可以访问对象。例如，美国军方将信息划分为绝密、秘密、机密或未分类。为了访问“秘密”类型的信息，工作人员必须具有对等或更高的安全许可级别。这种方案假定使用可信计算基（Trusted Computing Base，TCB）来对其控制下的所有主体和对象执行策略。

20 世纪 80 年代，美国国防部（Department of Defense，DoD）出版了第 1 本关于可信计算机系统的评估标准[28]（因出版物的封面颜色为橙色，我们称之为橙皮书），明确将强制访问控制定义为“基于对象中所包含的信息的敏感性（以标签表示）和主体访问这些敏感性信息的正式授权（即许可证）限制对象方法的一种方法”。定义基于 Bell－LaPadula 模型[29]，它影响了诸多计算机安全技术的发展。由于模型中使用了多个安全级别，因而将形成的系统称为多级安全系统。

Bell－LaPadula 模型主要通过“读取”或“写入”策略从根本上解决信息流控制问题。策略执行保证信息不会向下流动。有益的结果是能够防止特洛伊木马恶意软件入侵。考虑特权用户爱丽丝结束运行受感染程序的场景。不使用强制访问控制，相关进程可以访问爱丽丝所在组织中存储人员薪资信息的文件（其对手安迪无法访问该文件），并将其内容写入安迪可以访问的另一个文件（例如，文件名为“薪资 2”）中。使用强制访问控制，该进程不能以低于爱丽丝的安全级别写入薪资 2 文件中；如果薪资 2 的安全级别很高，则安迪无法读取该文件。

在 Bell－LaPadula 模型的启发下，人们开发出了 Biba 模型[30]。该模型涉及分类信息的完整性。具体来说，Biba 模型规定了“读取”和“写入”策略，这实际上由 Bell－LaPadula 模型禁止。基本原理是，对象的完整性只能与那些对其内容形成有贡献的对象中最不可靠对象的完整性一样。Bell－LaPadula 模型还激发了许多其他用于解决其局限性的开发活动。与云计算特别相关的是两大重要开发活动：类型强制访问控制和基于角色的访问控制（Role－Based Access Control，RBAC）。下面对其进行详细描述。

类型强制访问控制[31]是厄尔·波波特和理查德·凯恩开发的。根据本章参考文献［32］，这一想法是由验证多级安全系统满足敏感度水平必须准确包含在打印输出中的要求这一问题引发的。显而易见，解决方案组件涉及管道的使用。类型强制访问控制方法从根本上解决了 Biba 模型中无法实现管道的问题。考虑一种通过网络秘密发送数据的应用。在这种情况下，管道将包含一个准备原始数据的进程（P_1）、一个用于加密数据的中间进程（P_2）和一个处理网络传输的进程（P_3）。高可读策略可以确保数据流的完整性，但无法强制实现管道结构。换句话说，任何 P_2 可读取的数据，P_3 也可以进行读取。然而，可以忽略 P_2，且可以将明文数据传输到网络。

采用类型强制访问控制，每个对象都被分配一种类型属性，且每个主体都被分配一种域属性。主体是否可以访问对象由预先配置的被称为域定义表（Domain Definition Table，DDT）的集中式表格进行控制。从概念上讲，该表包括用于表示每种类型的行和用于表示每个域的列。行和列交点处的项指定了该域中的主体对该类型对象可能拥有的最高访问权限。实际上，该表是另一种访问矩阵。每当主体寻求访问对象时，都会对表格进行检查。如果所尝试的访问模式不在表中，则访问将被拒绝。后来，李·拜哲（L. Badger）等[33]将类型强制扩展为域型强制（Domain and Type Enforcement，DTE）。它根据文件层次结构中的位置，为访问控制配置和文件的隐含类型引入了高级语言。类型强制和域型强制比传统多级安全方案更为普遍。它们不仅可以实现信息流的机密性和完整性，而且还可以确保管道和安全性内核。

强制访问控制的著名实现方案是基于美国国家安全局（National Security Agency，NSA）Flask 框架



对 Linux 的 SELinux 内核扩展。除了多级安全和类型强制，SELinux 还支持多种类型的安全性。所谓的类型为控制增加了另一个维度。类型是根据给定策略拥有相同权限的一组主体。例如，类型分配和策略可能是两大主体：a 和 b，它们属于 X 类。两个主体 a 和 b 之间可以进行彼此通信，且能与 X'类中的主体进行通信，但 X 类中的主体无法与 Y 类中的主体进行通信。

就实现方案而言，类型可以由其各自的标识符或标签来表示。通过将主体与对应标签相关联，中心机构将为一个主体分配一种类型。对象执行的是相同的进程。在所有其他合适的策略中，基于类型的策略能提供一套额外的独立控制集。

SELinux 支持 sVirt 服务，旨在实施虚拟机隔离。使用 sVirt 服务，可以为每台虚拟机分配唯一的标签㊀。此外，可以将同一标签分配给与虚拟机相关的资源（如文件和设备）。通过禁止进程访问不同标签的资源可以确保虚拟机的隔离。如果虚拟机管理程序正确实现，则虚拟机可能无法访问另一台计算机的资源（如磁盘镜像文件）。但是，现实中不存在绝对安全和无漏洞的虚拟机管理程序。鉴于这种情况，sVirt 增加了额外的保护功能。

美国国家标准与技术研究院（NIST）牵头开发了基于角色的访问控制技术。本章参考文献［34］是一本关于 RBAC 的综合性专著，除了其他有用信息之外，还提供了 RBAC 详细的发展历史，并追溯到 1879 年收银机的发明。RBAC 根植于企业，其中的每个人都具有明确的功能（或角色），它反过来又定义了授权个人可访问的资源。根据本章参考文献［34］介绍，NIST 于 1992 年首先在商业和政府机构中开始研究访问控制，并得出技术方面存在严重差距的结论。特别是他们发现，基于最小特权原则，针对基于主体的安全策略和访问没有实现方案支持。

为了解决这些问题，本章参考文献［35］最早提出了 RBAC 模型，随后根据作者名字将该模型命名为 Ferraiolo – Kuhn 模型。该模型使用集合论结构形式化定义了角色层次结构、主体角色激活以及主体对象调解及其约束条件。RBAC 模型的核心是 3 条规则：①角色分配；②角色授权；③事务授权。第 1 条规则假设主体只有在分配角色后才能执行事务，然后根据第 2 条规则，必须为该主体授权。此外，根据第 3 条规则，必须对事务本身进行授权。换句话说，对任何资源的访问只能通过由一组许可定义的角色进行。与访问控制列表的情况正好相反，直接撤销主体的角色会导致主体无法对资源进行访问。该模型的一个重要特征体现在角色是分层而不是平面的（与访问控制列表组的情形一样），且角色可以继承。该特征简化了角色之间权限的聚合。例如，分配给初级经理的权限可以进行累积，并可将其分配给高级经理。该模型的另一个重要特征是可以使用限制条件（以根植于职责分离㊁的策略规则形式，并基于最小特权原则）来实现高级安全目标（如防止利益冲突）。例如，可以定义策略规则来阻止同时为用户分配或激活两个互相矛盾的角色（如系统管理员和审计员）。

按照原始建议，RBAC 研究在美国和国际上如雨后春笋般迅速增加，且其应用开始出现在理财、工作流管理和医疗保健等领域。到 2003 年底，业务已经对 ANSI RBAC 标准达成共识[36]。该标准已被广泛接受为国家标准制订的基础。例如，美国健康保险携带和责任法案（Health Insurance Portability and Accountability Act，HIPAA）具体规定了 RBAC 的使用。需要注意的是，在 RBAC 出现之前，人们将访问控制模型要么看作是强制性的，要么看作是自主的。RBAC 改变了这种观点。提供了一种灵活的策略框架，但未提供具体策略，它既支持自主访问控制，又支持强制访问控制[37]。

7.4.4 动态授权



动态授权是一项相对较新的发展成果。传统身份和访问管理系统可以处理事先设置的授权（但不能随时）。现在考虑以下源于网络的著名用例。爱丽丝将其照片存储在 TonVisage 网站上，该网站是由 TonVisage 公司运营的。访问这些照片是受密码保护的。只有爱丽丝知道其密码，她想以这种方式来保存密码。不过现在，爱丽丝想打印其照片（这些照片是在千载难逢的喜马拉雅山旅行中拍摄的），并制作一本实体相册。为了达到这一目标，她计划使用 PrintYerFace. com 提供的打印服务，该网站承诺以相

㊀ 实际上，标签是 SELinux 中主体或对象安全情境的一部分。除了“标签”（令人困惑的是，在 SELinux 中被称为“级别”），安全情境包含 3 个以上的字段：用户、角色和类型。

㊁ 职责分离是执行关键任务需要多个角色的原则，且无法将所有角色分配给一个人。RBAC 支持静态和动态职责分离。两者之间的关键区别在于：在动态职责分离中，约束条件是在运行时设置的。

对较低价格对爱丽丝的照片进行专业编辑，并将其打印在相册中漂亮的厚纸上。PrintYerFace 如何获取爱丽丝的照片？爱丽丝当然不想向任何人泄露其在 TonVisage 的登录名和密码。也许 PrintYerFace 会要求爱丽丝邮寄其照片。但这对爱丽丝来说是非常不方便的。对于 TonVisage 来说，根据公司每年增加 100 万用户的业务计划，提供照片所需的存储空间也将是一种负担。此外，还面临着商业竞争：例如，爱丽丝可以将照片贴在棍子上，然后走进当地打印店。常识要求爱丽丝尽可能简单地做事：她应当临时许可 PrintYerFace 在 TonVisage. com 上读取喜马拉雅山的照片。

为了支持用例，OAuth 社区努力于 2007 年开发了初始开放解决方案。这是一个实际存在的解决方案。实际上，所谓的 Web 2. 0 公司（如谷歌、美国在线、雅虎和 Flickr）都采用了太多太多的专有解决方案。后来，IETF 采纳了这种解决方案（称其为 OAuth 1. 0），并成为 OAuth 2. 0 的基础。OAuth 2. 0 是在特定特许工作组下开发的正式标准。IETF 中的 OAuth 标准化工作引起人们极大的兴趣。网络设备供应商和主要电信提供商很快加入 Web 2. 0 公司，这些公司积极参与了社区的标准化工作。随着时间的推移，出现了比打印照片问题更为严重的新应用，特别是云计算中虚拟机生命周期的自动化管理。

IETF 的第一个 OAuth 输出是与 OAuth 1. 0 有关的信息类 RFC。虽然不是官方标准，但它是作为实现规范而被使用的。RFC 与原始社区规范的主要区别在于通过在需要时强制采用 TLS 协议，它已经解决了一些安全问题，明确了时间戳和其他随机数的使用方法，并对加密签名进行了改进。

回到爱丽丝的用例上，图 7. 23 从概念上描述了 TonVisage 如何基于 OAuth 1. 0 从爱丽丝那里获取访问其照片的权限。从爱丽丝到 PrintYerFace 的初始请求（打印其相册）不是标准交换的一部分（因为请求本身不需要任何新的标准化），这就是为其分配一个零序列号的原因。在收到爱丽丝的请求后，PrintYerFace 反过来在消息 1 请求中访问 TonVisage 上的爱丽丝相册。现在，TonVisage 需要获取爱丽丝的许可，才能将该相册传给 PrintYerFace，这可以通过交换消息 2 和 3 来实现。

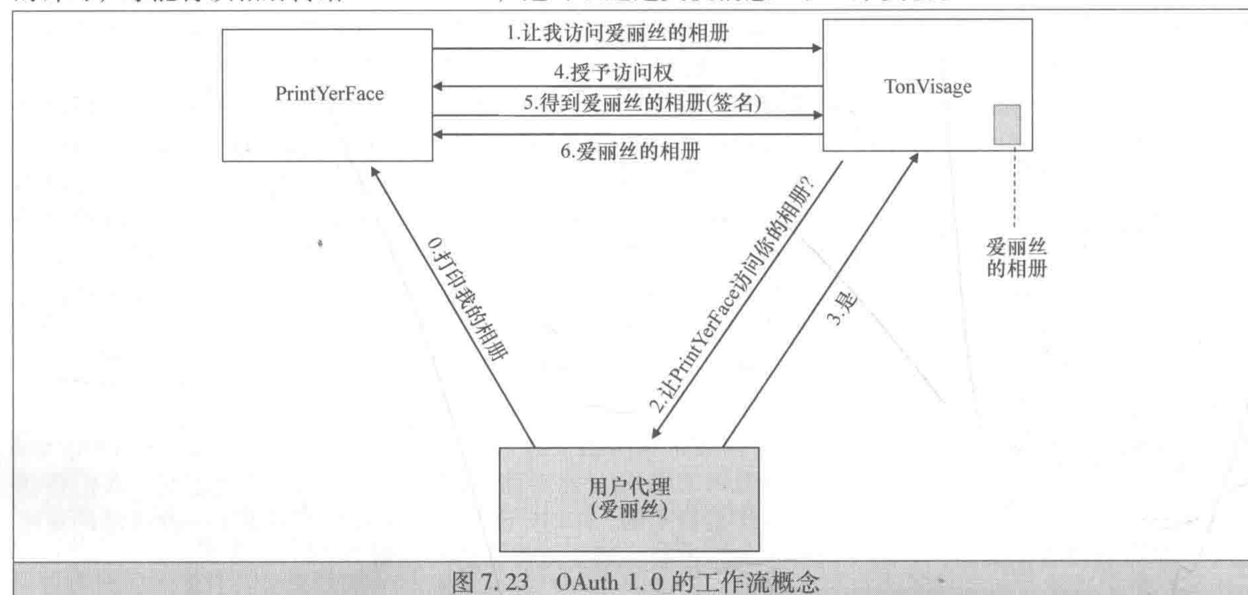


图 7. 23　OAuth 1. 0 的工作流概念

这里存在两个主要问题，就所涉及的认证主体而言：首先，TonVisage 需要理解消息 2 和 3 是与爱丽丝（而不是假冒她的人）对话的一部分；其次，TonVisage 必须知道它实际上是从 PrintYerFace 那里接收请求并向 PrintYerFace 发布信息。第 1 个问题是通过使用爱丽丝与 TonVisage 的密码来对其进行认证。第 2 个问题可通过 TonVisage 和 PrintYerFace 共享一组长期和短期的秘密密码来解决。长期共享的秘密支持 PrintYerFace 向 TonVisage 认证；短期共享秘密支持 PrintYerFace 证明它拥有合适的授权。短期秘密由 TonVisage 动态生成，并将其包含在消息 4 中返回给 PrintYerFace。使用长期和短期秘密的组合来对爱丽丝的相册请求进行签名。正如消息 5 和 6 所示的，PrintYerFace 可以最终得到相册。

然而，OAuth 1. 0 方案存在着缺点。首先，PrintYerFace（即 OAuth 1. 0 术语中的客户端）需要获取不同类型的长期和短期凭据，并使用其组合来生成数字签名。事实证明，该方案太复杂，Web 应用开发人员不容易实现。此外，OAuth 1. 0 还有太多需要实现的东西；该规范仍然警告粗糙实现方案可能会引起的一系列潜在灾难。一方面，协议不提供请求的机密性，因而窃听者可以看到这些请求。另外，



如果未对 TonVisage（即 OAuth 1.0 术语中的服务器）进行正确认证，则事务可能会被操纵。同时，为了计算签名，用于签名的共享秘密必须以明文形式提供给 OAuth 代码。遗憾的是，通过在服务器上以明文形式存储秘密，这一要求太容易实现。如果这样做，则突破服务器的攻击者可以通过伪装成攻击中获得共享秘密的客户端，来执行任何操作。

如果这还不够，则会出现更多潜在的问题，因为客户端主机上的实际 OAuth 代码通常属于第三方。如果服务器在认证中使用了多种因素，那么这一问题可以得到缓解，但它又不是协议规范的一部分，它只是“智能”服务器应该做的事情。知名的钓鱼攻击是另一种严重问题。只有那些知道他们正在访问何种网站的“聪明”用户才能克服这一问题。其他安全问题与秘密的长度及其生命周期有关。同时，由于没有机密性保证，窃听者可以收集经过认证的请求和签名，然后发起离线攻击来恢复秘密。

这些并非所有已知的安全问题，但现在应当清楚为什么 OAuth 1.0 需要认真地开展后续行动。新框架和协议不仅要解决安全问题，而且还要支持新用例，如虚拟机的生命周期自动化管理。新用例提出了新要求，特别是使用独立专用服务器来灵活处理认证和授权。考虑到这些需求，IETF RFC 6749 中规定的 OAuth 2.0 框架包括 4 个角色。除了 OAuth 1.0 使用的资源所有者（如爱丽丝）、客户端和服务器之外，还引入了与授权相关的另一种服务器。新服务器自然被称为授权服务器，而原始服务器被称为资源服务器。实际上，授权服务器的引入受到云计算的推动，因为云计算可能涉及依赖同一授权服务器的多台资源服务器。授权服务器通过统一资源标识符（Uniform Resource Identifier，URI）的服务端点来提供服务。

图 7.24 从概念上描述了同一用例的 OAuth 2.0 工作流。它与 OAuth 1.0 工作流类似，特别是如果 TonVisage 也可充当授权服务器时。主要区别在于 PrintYerFace 如何证明它拥有爱丽丝的授权。PrintYerFace 不像 OAuth 1.0 那样提供签名，而是出示授权服务器发行的访问令牌。令牌表示一组临时权限。同时，它可能并不限于特定客户端。将这种访问令牌称为承载令牌。合法承载令牌的持有者可以通过提供令牌来访问相关资源。这里，将承载令牌比作是音乐会门票。因此，承载令牌在运动和休息时均能得到适当保护，这一点是至关重要的。当然，让令牌的有效期尽可能短暂，有助于降低因信息泄露而导致的任何损失。

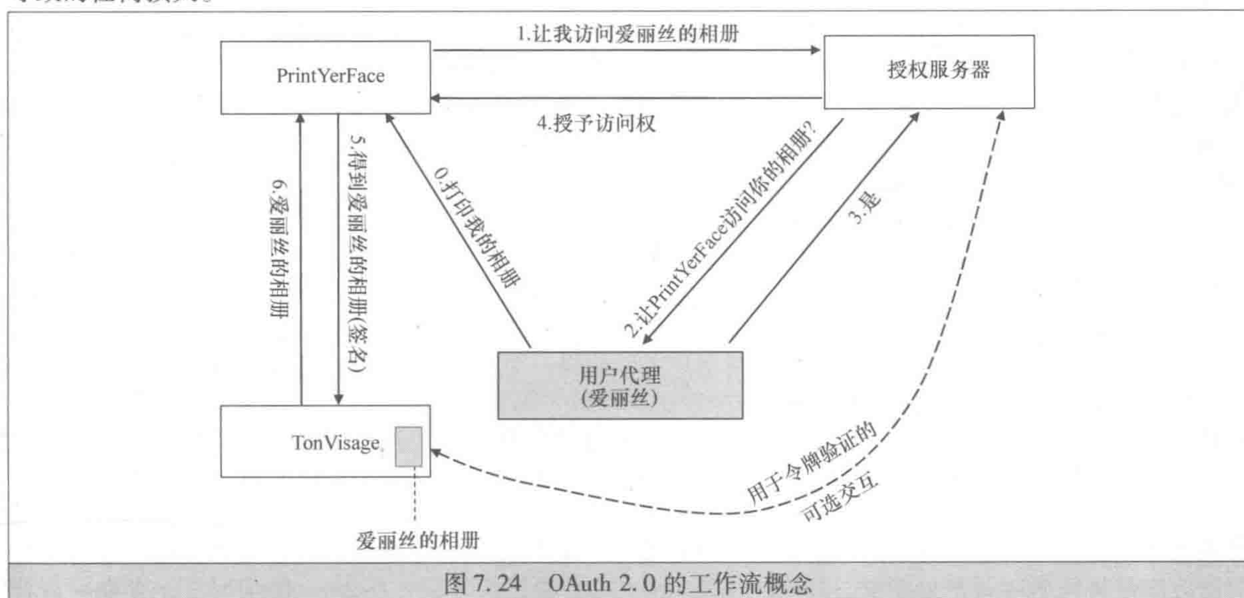


图 7.24 OAuth 2.0 的工作流概念

显而易见，在收到资源请求后，资源服务器必须验证所包含的访问令牌，并确保：①访问令牌尚未过期；②访问令牌的范围覆盖所请求的资源。资源服务器如何验证访问令牌超出了 OAuth 2.0 的范围。我们可以观察到，令牌验证实际上存在两种选择：

1）资源服务器。在这种情况下，令牌需要采用标准格式并捕获可验证的信息。依据的标准是来自 IETF 的 JSON Web 令牌（JSON Web Token，JWT）规范。JWT 表示是特别紧凑的，它允许在 HTTP 授权头或 URI 查询参数中承载如此表示的访问令牌。不推荐采纳后一种选项；URI 参数很可能被记录。该表示还支持对关键信息进行签名和加密。

2）授权服务器。在这种情况下，令牌格式由授权服务器决定，因为它负责访问令牌的发行和验证。授权服务器可以通过特定端点来提供令牌验证服务。

这两种选择都与云计算相关，稍后将在 OpenStack Keystone 个案研究中回顾。

7.4.5 联合身份 ★★★

联合身份最早在企业和网络行业中引入，它允许用户访问不同管理域中的应用，而无须使用与域有关的凭据或重新进行显式认证。这主要通过将身份管理组件与每种应用分离开来，并将其外包给一般身份提供商（Identity Provider，IdP）来实现。由此形成的分布式架构需要建立应用提供商和身份提供商之间的信任，以及用户与身份提供商之间的信任。然后，应用可以依靠身份提供商来验证用户的凭据，而用户反过来又可以享受更少凭据和单点登录带来的便利。在联合身份的术语中，用户被称为当事人（或请求人、主体），应用被称为依赖方。

这里，基本要求是多个服务提供商使用一组通用机制来发现用户、依赖方和身份提供商的身份。确保在这些实体之间甚至可能跨多个领域交换的敏感信息的安全（如机密性和完整性得到保护）也是至关重要的。

存在着支持联合身份的若干种机制。在附录 A 中，将回顾 3 种特别重要的机制：安全断言标记语言（SAML）、OpenID 和 OpenID Connect（这实际上是 OAuth 2.0 的一种应用）。

7.4.6 OpenStack Keystone（个案研究）★★★

Keystone 是 OpenStack 服务的守门员，能够提供集中式认证和授权。它既可以控制虚拟机用户，又可以控制 OpenStack 服务。Keystone 提供被称为身份服务和令牌服务的核心服务。

身份服务用于处理用户的基本认证和相关数据的管理。用户可以是个人或进程。基于密码的用户认证是本地支持的，但可以通过外部插件来使用其他认证方法。除了用户之外，Keystone 还支持域、项目、组和角色的结构。前 3 种结构与人员和资源的组织有关，域级别最高。因此，封装一组 OpenStack 资源的项目必须恰好属于一个域。类似地，由用户构成的组必须恰好属于一个域。角色结构是一个不同的故事。它代表一组权限，能够为访问控制提供依据。为了访问资源，必须为用户分配角色，且角色分配始终在项目或域上进行。Keystone 为角色使用单个命名空间。一个知名的角色是管理员。当涉及用户时，身份服务还支持联合身份，以允许组织重用其现有身份管理系统，从而节省了 Keystone 中现有组织用户的配置需求。Keystone 具有支持多种联合身份协议（如 SAML 2.0 和 OpenID Connect）的灵活性。目前，Keystone 已完成对 SAML 2.0 的支持。

令牌服务的核心是令牌的概念。图 7.25 给出了一种简化工作流来说明其功能。同时，考虑配置虚拟机的情况。在访问 Nova 或其他服务之前，Keystone 首先对用户进行验证。在认证成功后，为用户提供用于表示一组权限的临时令牌。从这一点开始，为了获得服务，用户（更确切地说，是用户代理 Horizon）将令牌封装在服务请求中。只有当令牌通过验证且令牌表示的权限符合服务的访问策略时，用户才能接收该服务。如果令牌已经过期，则用户可以获得一个新令牌，并尝试重新获取该服务。

如图 7.25 所示，Nova 要求 Keystone 对令牌进行验证，因为 Keystone 负责代码生成和令牌验证。这不是唯一的处理办法。稍后将讨论另一种方法和两种方法的折中。现在，需要注意的是，当要求 Keystone 验证用户令牌时，Nova（或任何其他服务）也需要提供自己的令牌（在图中用 tokenN 表示）进行认证。Nova 采取与终端用户相同的方式来获取令牌。它需要通过认证。但两者仍存在区别。Nova 无法像人类那样通过输入密码或其他类型的凭据进行交互。它只能获取存储在预设位置的凭据。最简单的解决方案是将凭据存储在文件中。的确，这是 OpenStack 的默认设置。在 Nova 的情况下，存在一份特定的配置文件，且用于验证的 Keystone 密码是明文。本书将在附录 A 中讨论存储密码的基本机制。该机制甚至向权限最高的管理员（即 root）屏蔽密码。但只有当认证过程是交互式时才有效。

为了对虚拟机进行配置，Nova 实际上需要来自其他 OpenStack 组件的服务。其他步骤如图 7.26 所示。Glance 保存了图像的注册表（包括块存储中特定图像的 URI），而 Swift 是将图像作为对象进行存储的对象存储。当 Nova 请求来自于另一种组件的服务时，它也会将用户令牌包含在内。这是非常必要的，因为每个 OpenStack 组件独立运行，以支持可扩展性。令牌支持 Glance 和 Swift 获知预期用户及其特权。这种基于令牌的方法具有重要的意义：它可以在不牺牲安全性的前提下提高可用性。在既不需



要重新进行显式认证又不泄露密码的情况下，用户可以一次登录并访问多种服务。最后，正如 Nova 先前所做的那样，Glance 和 Swift 必须分别要求 Keystone 验证令牌，且在这样做时，必须提供自己的令牌（分别表示为 tokenG 和 tokenS）。

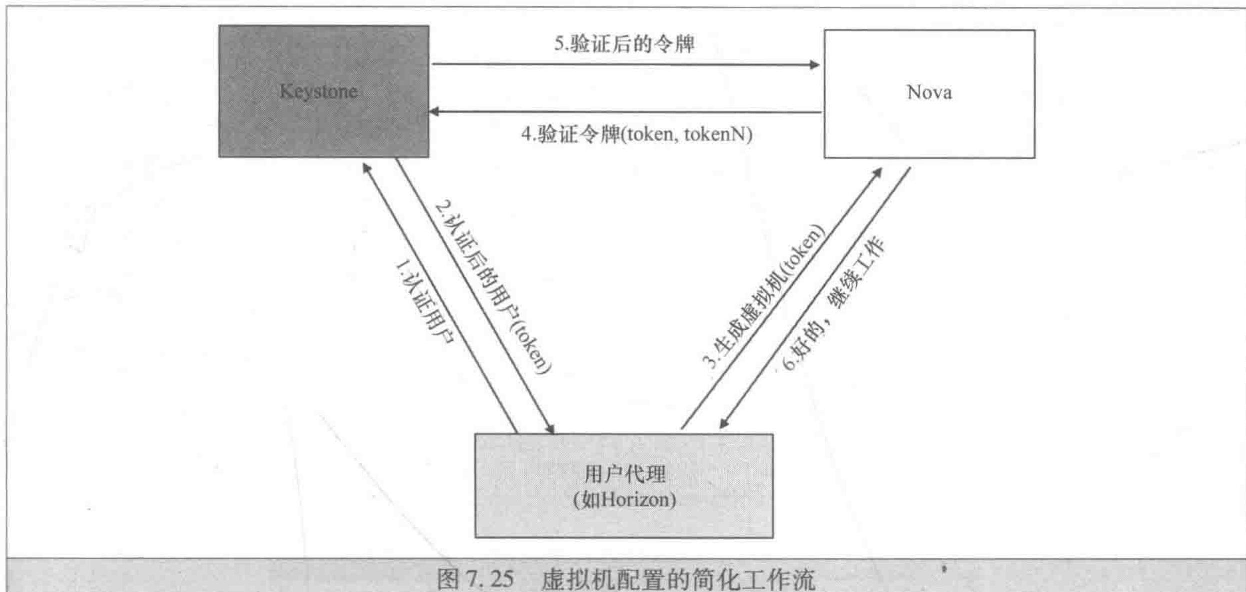


图 7.25 虚拟机配置的简化工作流

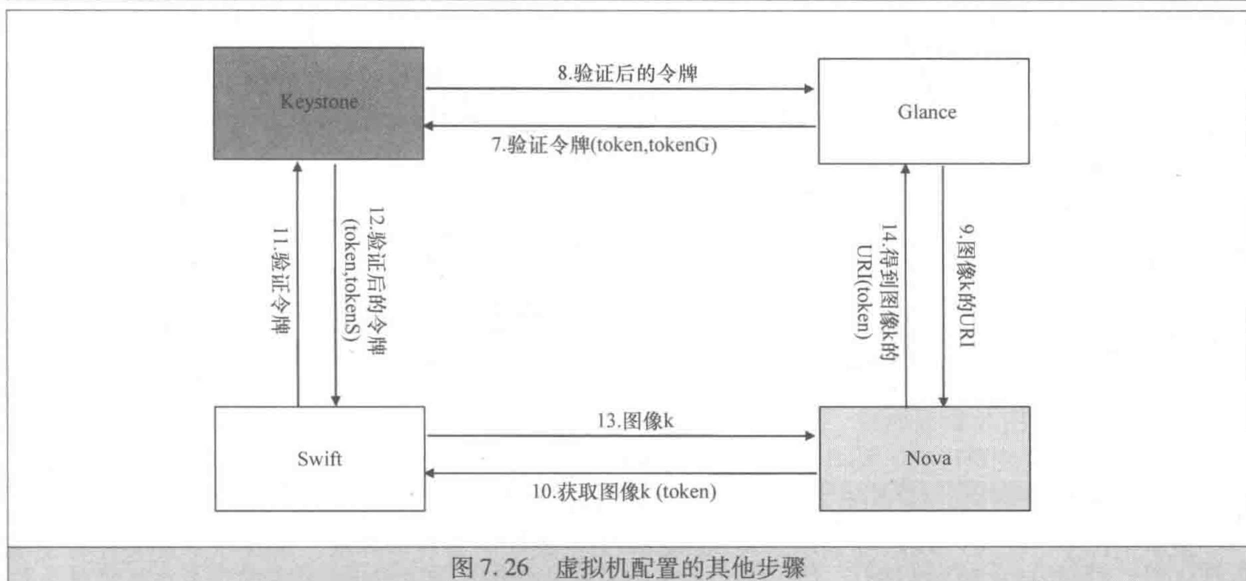


图 7.26 虚拟机配置的其他步骤

令牌通常捕获发布时间、有效期和用户信息。它还可以定义范围——与域有关的角色信息。如果令牌不包含此类信息，则将其称为“无作用域”。此类令牌对于资源访问是无用的，但它可以用作发现可访问项目的垫脚石，然后为其交换一个作用域令牌。令牌甚至可以捕获授予用户的服务目录。

迄今为止，应该澄清的是，Keystone 令牌属于承载令牌。作为服务访问的基础，必须确保令牌免受信息泄露、伪造和变更的威胁。信息泄露保护是一项大课题，涉及通信与信息安全等综合类问题。防伪和完整性保护是不同的事情，易受到令牌结构和格式的影响。虽然可以通过外部模块来使用自定义令牌，但 Keystone 本身支持 Juno 迄今为止发布的 3 种类型令牌，分别命名为通用唯一标识符（Universally Unique Identifier，UUID）、公钥基础设施（PKI）和压缩版 PKI。

UUID 令牌是随机生成的字符串，用作对存储在持久性令牌数据库中的某条信息的引用。它本身没有任何意义。服务必须调用 Keystone 来验证接收到的任何 UUID 令牌。实际上，这是在前面的工作流中所看到的。如果数据库中存在一种尚未过期的匹配令牌，则 UUID 令牌是有效的。只有 Keystone 可以验证令牌的约束条件使其成为潜在的瓶颈。然而，也有好的一面。UUID 令牌的尺寸很小且固定（即 128

位)，因而在实践中它将永远不会导致 API 调用失败[⊖]。此外，伪造有效的 UUID 令牌相对较难。Keystone 使用 IETF RFC 4122 中规定的 UUID 版本 4。这样的 UUID 包含生成的 122 位伪随机数。这意味着创建已在数据库中的 UUID 的概率是 2^{-122}。最后，改变现有令牌会导致新令牌的产生。同时，新令牌在数据库中存在的概率是 2^{-122}。

与 UUID 令牌不同，PKI 令牌不仅指向数据结构的指针，而且还指数据结构本身。该结构存储用户身份和授权信息。为了防止对内容进行修改，数据结构包含有数字签名。这种令牌的直接好处是可以通过服务端点进行验证。因此，关于 UUID 令牌的瓶颈和可扩展性的担忧已不复存在。具体来说，PKI 令牌是一种密码消息语法（Cryptographic Message Syntax，CMS）字符串（使用 Base64 编码方式）。该令牌包含经过数字签名的数据块。数据的具体情况与情境有关。因此，令牌大小是变化的。作为实例，图 7.27 给出了签名之前的 JSON 令牌。需要注意的是，它包括用于用户认证的方法列表中的强制性信息。此外，为了支持拥有令牌的认证用户能够得到不同范围的另一个令牌，也可以将令牌看作是一种认证方法。当认证用户执行此操作时，认证方法列表将包括“密码”和“令牌”。

```
{
    "token": {
        "expires_at": "2017-05-27T22:52:58.852167Z",
        "issued_at": "2017-05-27T21:52:58.852167Z",
        "methods": ["password"],
        "domain": {
                "id": "3a5140aecd974bf08041328b53a62458",
                "name": "Wonderland"
        },
        "roles": [{
                "id": "9fe2ff9ee4384b1894a90878d3e92bab",
                "name": "admin"}
        ],
        "user": {
            "domain": {
                "id": "3a5140aecd974bf08041328b53a62458",
                "name": "Wonderland"
            },
            "id": "3ec3164f750146be97f21559ee4d9c51",
            "name": "Alice"
        }
    }
}
```

图 7.27　令牌（无符号）实例

默认情况下，RSA－SHA256 是用于在 Keystone 中生成数字签名的算法。这意味着需要计算使用 SHA256 算法对数据块摘要的签名，并通过使用 RSA 私钥对摘要进行加密来计算签名。除了其他标准[⊖]，如果签名有效，则可判定 PKI 令牌合法的。为了方便另一服务端点的签名验证，Keystone 支持通过其 API 来获取签名和其他相关证书。

PKI 令牌的特征之一是它的尺寸较大。令牌在尺寸上可以轻而易举地达到数千字节。特别是如果令牌包含用户可访问的服务目录，则其大小可能达到 HTTP 报头的实际极限值，并中断该操作。为了减小令牌尺寸，Keystone 支持压缩。

遗憾的是，无法保证压缩 PKI 令牌的大小始终保持在 HTTP 报头极限值之内。由于 UUID 和 PKI 令牌之间的折中，Keystone 中的默认令牌类型不是一种解决方案。最初是 UUID 令牌，在 Grizzly 版本中变

⊖ OpenStack API 服务器实现是基于现有开源 HTTP 服务器软件构建的。虽然 HTTP 标准对报头字段的大小没有限制，但由于操作和安全性原因，此类软件也是这样进行处理的。例如，Apache 2.4 服务器默认将报头字段的大小限制为 8190 字节。

⊖ 其他令牌验证标准是有效期未到，且令牌尚未被撤销。



成 PKI 令牌，从 Juno 版本开始又变回 UUID 令牌[1]。

迄今为止，已经知道了 Keystone 令牌如何支持单点登录和基本授权。这些令牌非常有用，且生命周期比较短暂（根据设计）。当所讨论行动的最后期限未知时，令牌可能会过期（且在现实生活中实际上真地会过期）。想象一下当现有服务器的 CPU 利用率达到 60% 时启动新服务器的任务。根据客户端的活动，这可能会在数分钟、数年内发生，甚至永远都不会发生。

因此，需要一些其他机制来动态获取令牌。为此，Keystone 引入了信任结构来捕获谁来授权、谁获得授权、授权范围和持续时间以及其他相关信息。

信任与授权书不同。授权实体生成授权书并将其交给被授权方。现在，被授权方提供用于获取令牌的信任。这种配置的基本思路是令牌是短暂的，但信任是长期的。

按照 Keystone 的说法，将授权实体称为委托人，被授权实体称为受托人。只有委托人才能创建信任关系。基于信任关系，受托人（但没有其他人）可以得到新令牌，以执行委派的任务。令牌将具有与信任相同的作用范围。将这种令牌称为信任令牌。如果不是为了携带信任相关信息的附加数据块，则其结构与标准令牌的结构相同。当然，在将信任用于获取令牌时，它可能不会过期。

为了探讨信任的工作原理，仔细研究一下用于新堆栈运行的工作流的相关部分。图 7.28 给出了一种极大简化的工作流，用于实例化模板以提供堆栈，并在运行堆栈过载时进行自动扩展。

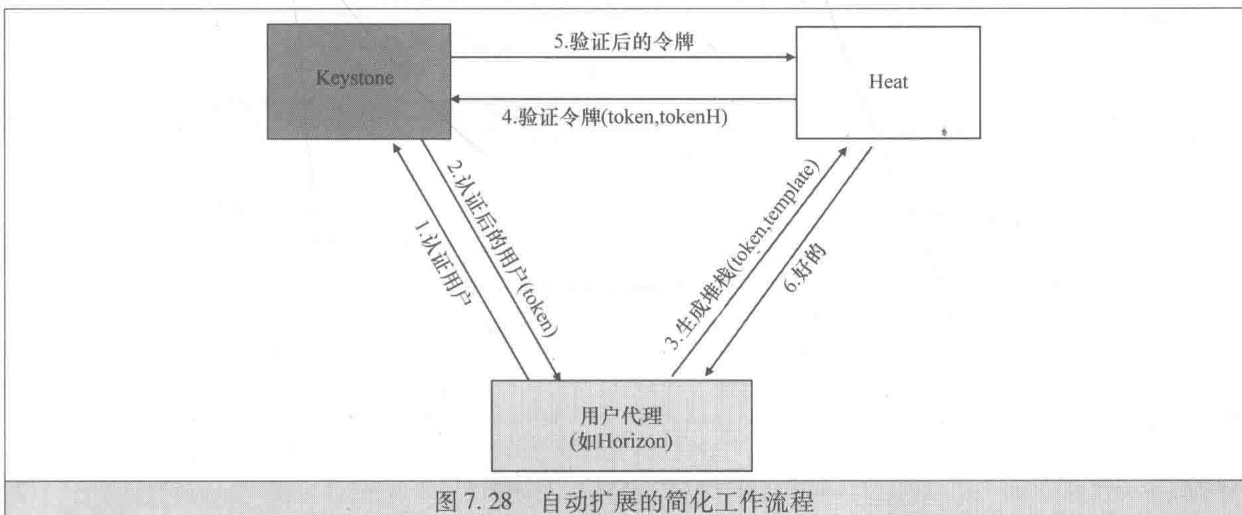


图 7.28　自动扩展的简化工作流程

当不涉及 Heat 时，步骤 1 ~ 5 基本上与先前相同。在步骤 5 之后，将发生一系列重头戏来启动第一个堆栈。这里，将重点介绍用于为自动化搭建舞台所需的其他步骤。这些步骤如图 7.29 所示。首先，设置一种告警（步骤 5a ~ 5d）；然后，创建一种信任关系，以支持 Heat（充当服务用户）拥有爱丽丝的授权（步骤 5e ~ 5f）。完成这些步骤之后，Heat 处于监控模式，等待告警在 Ceilometer 上出现。最后，告警通知在步骤 7 到达[2]。收到通知后，Heat 继续获取信任令牌，如步骤 8 所示。这里，Heat 需要提供自己的令牌来证明它是信任关系中可以识别的受托人。认证成功后，Heat 接收到一个信任令牌，它支持 Heat 代表爱丽丝启动一台新虚拟机。

图 7.30 给出了一个信任实例。除了“假冒”字段之外，该结构几乎是不解自明的。如果设置为假，则基于该信任的信任令牌的用户字段将表示受托人。否则，用户字段将表示该委托人。换句话说，受托人在提供用于验证的信任令牌时，假冒委托人。在实例中，信任令牌将爱丽丝作为用户的情况就是这样。在给定用于提供用户堆栈自动生命周期管理目标的前提下，Keystone 中默认支持假冒。为达到同一目标，另两个字段——expires_at 和 remaining_uses 也设置为默认值。如果未加指定，则 expires_ at 字段将拥有该信任有效直到明确撤销的效果。类似地，如果未加指定，则 remaining_uses 字段将具有这种效果：可以使用该信任来无限次地获得令牌。最后，一旦信任创建则不可改变。为了更新信任关系，委托人将删除旧信任，并创建新信任。在删除旧令牌后，由其衍生的任何令牌将被撤销。如果委托人

[1] OpenStack 版本按字母顺序进行命名。
[2] 虚线用于反映不使用 Keystone 进行认证的独立通知机制的事实。

失去任何委托权限，则信任无效。

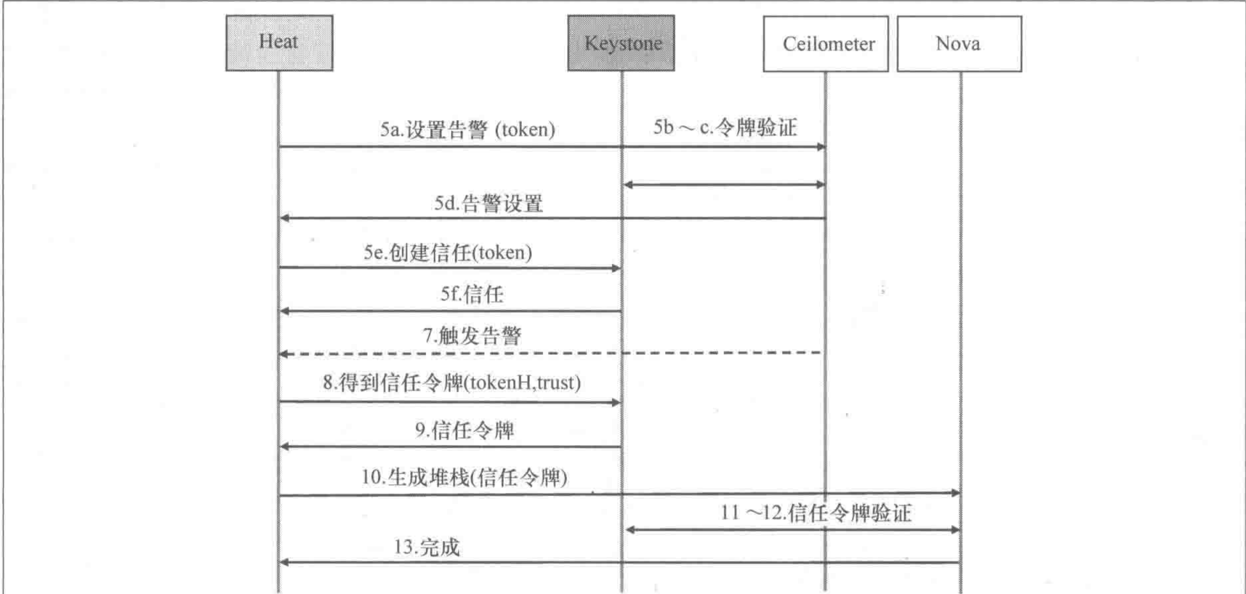


图 7.29　自动扩展的其他步骤

```
{
    "trust": {
        "expires_at": "2016-05-27T21:52:58.852167Z",
        "id": "c703057be878458588961ce9a0ce686b",
        "impersonation": true,
        "project_id": "fb49a0ecd60c4d2092643b4cfe272106",
        "remaining_uses": null,
        "roles": [{
        "id": "9fe2f79ee4384b1894a9083bd3e92bab",
        "name": "admin"}
        ],
        "trustee_user_id": "29beb2f1567642eb810b042b6719ea88",
        "trustor_user_id": "3ec3164f750146be97f21559ee4d9c51"
    }
}
```

图 7.30　信任实例

对联合身份的支持会导致新的认证工作流出现。当接收到认证请求时，Keystone 不是处理请求本身，而是将其重定向到外部身份提供商。在接收到重定向请求后，身份提供商执行用于认证用户的步骤，然后将结果重定向到 Keystone。如果证明用户是可信的，则 Keystone 会生成一个无作用域令牌。用户可以根据该令牌找到可访问的项目，并获得具有恰当作用范围的另一个令牌。针对联合用户生成的令牌是不同的——它们携带与联合相关的信息，如身份提供商的名称、联合身份协议和关联组。

总之，令牌服务不仅支持令牌的生成、撤销和验证，而且还支持信任的生成、撤销和验证。作为集中式服务，它受到 OAuth 2.0 的影响，且可以直接映射到 OAuth 2.0 的授权服务器。令牌类似金钱的特征（令牌的任何载体都可以使用它）以及使用 JSON 进行令牌描述与此相似。然而，Keystone 令牌服务拥有自己的协议（以 REST API 的形式），因为其目标是非常不同的用例。本质上，用例包括两种角色（即用户和云基础设施服务提供商），其自动化程度高，但互动性低。因此，用于委托的不同用户满意度模型已经足够用了——用户通过用来获得服务的认证后，就意味着用户同意委托。



参考文献

[1] Blatter, A. (1997). Instrumentation and Orchestration, 2nd edn. Shirmer, Boston, MA.

[2] Hogan, M., Liu, F., Sokol, A., and Tong, J. (2011) NIST Cloud Computing Standards Roadmap—Version 1. 0. Special Publication 500 - 291. National Institute of Standards and Technology, US Department of Commerce, Gaithersburg, Maryland.

[3] Liu, F., Tong, J., Mao, J., et al. (2011) NIST Cloud Computing Reference Architecture. Special Publication 500 - 292. National Institute of Standards and Technology, US Department of Commerce, Gaithersburg, Maryland.

[4] Stroustrup, B. (2013) The C + + Programming Language, 4th edn. Addison - Wesley, New York.

[5] Birman, K. P. (2012) Guide to Reliable Distributed Systems: Building High - Assurance Applications and Cloud - Hosted Services. Springer - Verlag, London.

[6] Lapierre, M. (1999) The TINA Book: A Co - operative Solution for a Competitive World. Prentice - Hall, Englewood Cliffs, NJ.

[7] Thompson, M., Belshe, M., and Peon, R. (2014) Hypertext Transfer Protocol version 2. Work in progress. https://tools.ietf.org/html/draft - ietf - httpbis - http2 - 12 IETF.

[8] Erl, T. (2005) Service - Oriented Architecture: Concepts, Technology, and Design. Prentice - Hall, Englewood Cliffs, NJ.

[9] Culler, D. E. (1986) Data Flow Architectures. MIT Technical Memorandum MIT/LCS/TM - 294, February 12. http://csg. csail. mit. edu/pubs/memos/Memo - 261 - 1/Memo - 261 - 2. pdf.

[10] Slutsman, L., Lu, H., Kaplan, M. P., and Faynberg, I. (1994) Achieving platform - independence of service creation through the application - oriented parsing language. Proceedings of the IEEE IN' 94 Workshop, Heidelberg, Germany, pp. 549 - 561.

[11] Amin, K., Hategan, M., von Laszewski, G., et al. (2004) GridAnt: A client - controllable grid workflow system. 37th Hawaii International Conference on System Science, pp. 210 - 220. (Also available in an Argonne National Laboratory preprint: ANL/MCS - P1098 - 1003 at www. mcs. anl. gov/papers/P109Apdf.)

[12] Kalenkova, A. (2012) An algorithm of automatic workflow optimization. Programming and Computer Software, 38 (1), 43 - 56. Springer, New York.

[13] Rey, R. F. and Members of the Technical Staff of AT&T Bell Laboratories (1983) Engineering and Operations in the Bell System, 2nd edn. AT&T Bell Laboratories, Murray Hill, NJ.

[14] Ebner, G. C., Lybarger, T. K., and Coville, P. (1991) AT&T and TÉLÉSYSTÉMES partnering in France. AT&T Technical Journal, 71 (5), 45 - 56.

[15] International Telecommunication Union (1998) International Standard 9596 - 1, ITU - T Recommendation X. 711: Information Technology—Open Systems Interconnection—Common Management Information Protocol: Specification.

[16] Dick, K. and Shin, B. (2001) Implementation of the Telecom Management Network (TMN) at WorldCom—Strategic Information Systems Methodology Focus. Journal of Systems Integration, 10 (4), 329 - 354.

[17] Schneiderman, A. and Casati, A. (2008) Fixed Mobile Convergence. McGraw - Hill, New York.

[18] Anderson, T. W., Busschbach, P., Faynberg, I., et al. (2007) The emerging resource and admission control function standards and their application to the new triple - play services. Bell Labs Technical Journal, 12, 5 - 21.

[19] International Telecommunication Union (2011) ITU - T Recommendation Labs.: Resource and Admission Control Functions in Next Generation Networks, Geneva.

[20] Camarillo, G. and Garcia - Martin, M. A. (2008) The 3G IP Multimedia Subsystem: Merging the Internet and the Cellular Worlds, 3rd edn. John Wiley & Sons, Inc., Hoboken.

[21] European Telecommunications Standards Institute (2005) Resource and Admission Control Subsystem (RACS), Functional Architecture. ETSI ES 282 003, v. 1. 6. 8, December. www. etsi. org/services_ products/freestandard/home. htm.

[22] Wayner, P. (2013) Puppet or Chef: The configuration management dilemma. Network World. www. infoworld. com/article/2614204/data - center/puppet - or - chef - the - configuration - management - dilemma. html.

[23] Rivest, R. L., Shamir, A., and Adleman, L. (1978) A method for obtaining digital signatures and public - key cryptosystems. Communications of the ACM, 21 (2), 120 - 126.

[24] Diffie, W. and Hellman, M. E. (1976) New directions in cryptography. IEEE Transactions on Information Theory,

22 (6), 644 - 654.

[25] Kohnfelder, L. M. (1978) Towards a practical public - key cryptosystem. B. S. thesis, Massachusetts Institute of Technology.

[26] International Telecommunication Union (2012) Information Technology—Open Systems Interconnection—The Directory: Public - Key and Attribute Certificate Frameworks. ITU - T Recommendation X. 509, December. www. itu. int.

[27] Bishop, M. (2014) Mathematical models of computer security. In Bosworth, S., Kabay, M. E., and Whyne, E. (eds), Computer Security Handbook, 6th edn. John Wiley & Sons Inc., Hoboken, Chapter 9.

[28] Brand, S. L. (1985) DoD 5200. 28 - STD Department of Defense Trusted Computer System Evaluation Criteria (Orange Book), National Computer Security Center, pp. 1 - 94.

[29] La Padula, L. J. and Bell, D. E. (1973) Secure Computer Systems: Mathematical Foundations. MTR - 2547 - VOL - 1, Mitre Corporation, Bedford, MA.

[30] Biba, K. J. (1977) Integrity Considerations for Secure Computer Systems. MTR - 3153 - REV - 1, Mitre Corporation, Bedford, MA.

[31] Boebert, W. E. and Kain, R. Y. (1985) A practical alternative to hierarchical integrity policies. Proceedings of the 8th National Computer Security Conference, Gaithersburg, MD.

[32] Boebert, W. E. and Kain, R. Y. (1996) A further note on the confinement problem. IEEE Security Technology' 96, 30th Annual 1996 International Carnahan Conference, Lexington, Kentucky.

[33] Badger, L., Sterne, D. F., Sherman, D. L., et al. (1995) Practical domain and type enforcement for UNIX. Proceedings of IEEE Symposium on Security and Privacy, Okland, CA.

[34] Ferraiolo, D. D., Kuhn, R., and Chandramouli, R. (2003) Role - based Access Control. Artech House, Boston, MA.

[35] David, F. and Kuhn, R. (1992) Role - based access controls. Proceedings of 15th NIST/NCSC National Computer Security Conference, Baltimore, MD.

[36] American National Standards Institute (2004) Role Based Access Control. ANSI INCITS 359 - 2004, February, New York, NY.

[37] Bonneau Osborn, S., Sandhu, R., and Munawer, Q. (2000) Configuring role - based access control to enforce mandatory and discretionary access control policies. ACM Transactions on Information and System Security (TISSEC), 3 (2), 85 - 106.

附录A 精选专题

本附录的目的是提供更多与前面引用主题有关的信息：

1）第一个专题关注的是IETF制定的有关业务和管理标准。

2）第二个专题介绍了一般建模语言的发展，特别是TOSCA标准。

3）第三个专题介绍了万维网（World Wide Web，WWW）架构中的REST API的概念。

4）第四个专题涵盖了启用认证和访问管理的基本机制。

A.1 IETF业务与管理标准

这里介绍的标准主要是简单网络管理协议（Simple Network Management Protocol，SNMP）、公共开放策略服务（Common Open Policy Service，COPS）和网络配置（Network Configuration，NETCONF）协议。

A.1.1 SNMP ★★★

互联网的OSI网络管理方法既没有被忽视，也没有被完全放弃。事实上，人们仅将其简化并限制在较小的具体问题上。到目前为止，网络管理的主要目标是监控，而不是配置管理。正如即将看到的那样，SNMP的使用不够普遍；而在配置管理方面，需要一套新的协议。另外，在SNMPv3之前，人们还没有系统地研究访问管理的内容（以及网络管理的整体安全性）。

1990年IETF RFC 1155定义了管理信息结构（Structure of Management Information，SMI）。与OSI一样，MIB使用ASN.1定义，但ASN.1的使用受到限制，相关原因将在稍后解释。通过对象标识符，人们已经为每种类型的对象分配了一个根据其编码指定的名称。根据管理策略对对象标识符进行分配。

对象标识符是通过全局命名树结构的路径[⊖]。边缘由Unicode字符串标记，Unicode字符串反过来又被编码成整数。每个边缘指示相应的管理域（可随着树被遍历而改变）。图A.1描述了具有3个边缘的树的根，它们分别由ITU-T（0）、ISO（1）和ISO与ITU-T（2）共同管理。

在ISO管理下，存在国际组织（3）的管理空间，它为美国国防部（US Department of Defense，US DoD）提供了相应的空间。其余的SMI路径列在该图的右手侧。“互联网（1.3.6.1）”子树被保留用于互联网编号分配机构（Internet Assigned Number Authority，IANA）管理的对象。

为了允许独立的对象定义，定义了“专用空间（1.3.6.1.4）”，将1.3.6.1.4.1的空间分配给企业对象。

一般，所有的标识符都是通过OBJECT IDENTIFIER结构递归定义的——通过引用之前定义的结构。因此，可以写为

```
internet OBJECT IDENTIFIER ::= {iso org (3) dod (6) 1};
private  OBJECT IDENTIFIER ::= {internet 4}; and
enterprise OBJECT IDENTIFIER ::= {private 1}.
```

RFC 1155只允许原始（非聚合）的ASN.1类型[⊖]。它限制了列表和表格构造函数类型的使用，并定义了很多的应用程序类型（例如，网络地址、IP地址、32位的计数器和0.01s间隔的定时器等）。RFC 1155进一步定义了MIB的格式。后者是一个五元组，包括：

⊖ RFC1155在描述这种树形结构方面有些不准确（并且有些过时），因此，使用ITU-T X.660建议书中的相关描述内容，其中包含了所有顶级的域名空间定义。

⊖ 它们是NULL、INTEGER、OCTET STRING和OBJECT IDENTIFIER。

根

2“Joint−ISO−ITU−T”(joint−iso−itu−t)

0“ITU−T”(itu−t 或ccitt)

1“ISO”(iso)

ISO

3“ORG”(org)

国际组织

6“DoD”(dod)

US DoD

1.3.6.1 internet
1.3.6.1.1 directory
1.3.6.1.2 mgmt
1.3.6.1.2.1 mib-2
1.3.6.1.2.1.2.2.1.3 ifType
1.3.6.1.2.1.10 transmission
1.3.6.1.2.1.10.23 transmissionppp
1.3.6.1.2.1.27 application
1.3.6.1.2.1.28 mta
1.3.6.1.2.2 pib
1.3.6.1.3 experimental
1.3.6.1.4 private
1.3.6.1.4.1 enterprise
1.3.6.1.5 security
1.3.6.1.6 SNMPv2
1.3.6.1.6.1 snmpDomains
1.3.6.1.6.2 snmpProxys
1.3.6.1.6.3 snmpModules
1.3.6.1.7 mail
1.3.6.1.8 features

源引自:ITU−T建议X.660|ISO/IEC 9834−1
IETF RFC 1155

图 A. 1　SMI ASN. 1 对象标识符树

1）对象名称（与 OBJECT IDENTIFIER 一起的文本 OBJECT DESCRIPTOR）。

2）对象结构规则，称为语法，必须解析为允许的结构类型。

3）对象类型“语义”的定义，该定义是人可读的，而不是面向机器的，用于确保“在所有机器上含义一致”。

4）访问（只读、读写、只写或不可访问[㊀]）。

5）状态（强制、可选或过时的）。

RFC 3411 中列出了 SNMP 的框架架构，它是互联网标准 62 的一部分。SNMP 系统通过在 SNMP 实体之间交换 SNMP 消息来实现。这些实体中至少有一个是管理者；其他实体存在于设备或网元中。在其更一般的形式中，实体的架构如图 A. 2 所示。



图 A. 2　SNMP 实体（引自 RFC 3411）

㊀ 不可访问的意思是“不可访问 SNMP”。一个重要的例子是加密密钥，必须对其进行保密。SNMPv3 模型（RFC 3414）里定义了多个这样的对象，但是它们都不可访问 SNMPv3 协议。



实体具有 SNMP 引擎和应用程序。引擎执行协议并且还负责安全服务，特别是机密性和认证。在一个管理域中，每个引擎（因此每个实体）都是唯一的，它被分配了一个名称，即 snmpEngineID。

该引擎由调度器、消息处理子系统和访问控制子系统组成。

调度器支持（并发）多个协议版本，但为 SNMP 应用程序提供单个抽象接口。

消息处理子系统[一]依赖于特定域 SNMP 版本的模块。

最后，安全子系统根据给定的模型提供同名服务。该模型以其所防护的威胁和其采用的协议为特点。只有基于用户的模型[二]才被明确提到，但该计划是允许所有的模型都可以作为插件。

访问控制子系统提供授权服务（因此，可以被认为是安全子系统的一部分）。与安全子系统类似，访问控制子系统依赖于一个或多个特定的访问控制模型。RFC 3415 是 STD 62 的另一部分，指定了基于视图的访问控制模型，该模型决定了群组的访问权限，代表具有相同访问权限的零个或多个对象。正如该标准所述，“对于由 contextName 标识的特定上下文，由 groupName 标识的群组可以使用特定的 securityModel（安全模型）和 securityLevel（安全等级）进行访问，该群组的访问权限针对 read - view（读视图）、write - view（写视图）和 notify - view（通知视图）指定”。每个视图均表示一组被授权其相应动作的对象实例（即在通知中读取对象或写入对象，或者发送对象）。

就 SNMP 实体中的应用程序而言，它们包括监视和操纵数据管理的命令生成器、对外提供数据管理功能的命令响应器、发起通知消息的通知发起器、处理异步消息的通知接收器，以及仅向接收方转发消息的代理转发器。这些应用程序都是预先定义好的，但其并没有排除其他的应用程序，因此该架构留有地方可以用来插入其他应用程序。

A.1.2 COPS ★★★

COPS 在 RFC 2748 中规定，但是这项工作的上下文已经在 RFC 2753 中进行了制定，它提供了开发的动机、描述了相关的术语，并以其他方式规定了框架。下面将简要介绍这个上下文内容。首先，该框架的目标是描述在 QoS 准入控制决策上基于策略控制的执行情况，其中主要关注 RSVP 并将其作为示例[三]。

该框架的目标是实现对抢占、各种策略风格、监管和计费的支持。这里的抢占意味着移除之前授予的资源以适应新的请求的能力[四]。就政策风格而言，那些需要得到支持的包括基于提供商定义的相对优先级概念的双边和多边服务协议和政策。通过收集资源使用和访问数据来实现对监控和计费的支持。该框架还规定了当 PDP 发生故障或无法达到的情况下容错和恢复方面的要求。

图 A.3 展示了策略控制架构。其主要组件是 PDP 和策略执行点（Policy Enforcement Point，PEP）。

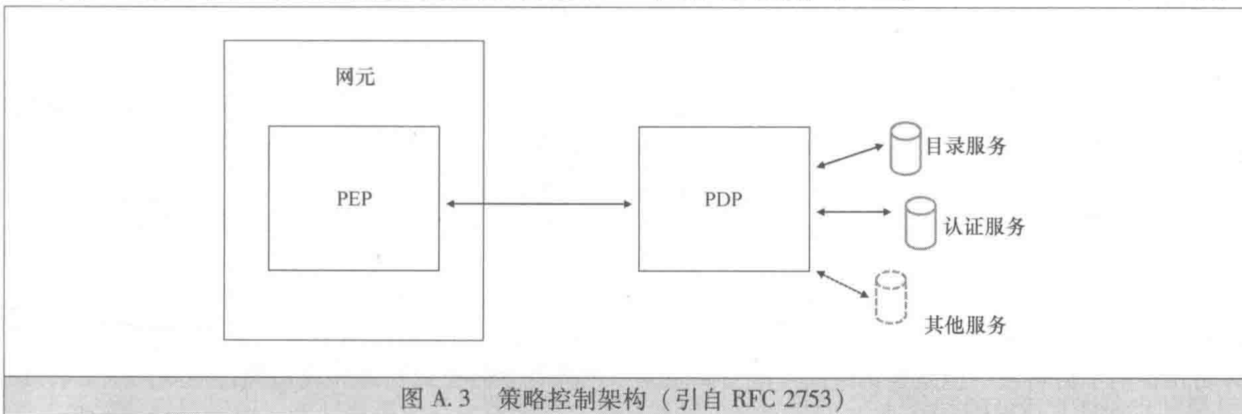


图 A.3 策略控制架构（引自 RFC 2753）

[一] 子系统这个词指的是一种全面的抽象机制。当被设计用来提供一组特定的服务时，它被称为模型。通过实现，可以将模型转化为真实的实体。

[二] 详细描述内容请参见 STD 62 RFC 3414。该文档的内容是独立且全面的，易于读者参阅。

[三] 在本节后面的内容中，将看到 ETSI、ITU - T 和 3GPP 如何进一步扩展该框架，使其不仅包含其他的 QoS 资源（例如，diffserv 代码点），还包括运营商控制下的其他所有资源：专用 IP 地址、端口号、NAT 打洞等。

[四] 这比拆东墙补西墙微妙得多。据本书英文原版作者的猜测，此处的目的是在极端负载条件下优化网络（和服务提供）。抢占还可以使我们能够处理资源管理的痼疾——“死锁问题”。

其中，PDP 进行基于策略的决策；PEP 查询 PDP 并确保后者的指令被执行。PEP 是执行准入控制网元的一部分。图 A.3 描述了 PDP 位于网元之外的一般情况。然而，PEP 和 PDP 可以同时位于同一个物理“盒子”内。为此，该框架允许同时使用处于共同位置的 PDP 模块［称为本地 PDP（Local PDP，LP-DP)］和远程 PDP 模块。

当 PDP 作用于事件（例如，决定是否接纳给定的分组）时，通常会触发 PEP 与 PDP 之间的交互。PEP 本身也可以通过发送通知来触发这种交互。在前一种情况下，PEP 需要查询 PDP，从而提供准入控制信息。其中后者可以是 flowspec、所请求的带宽量或触发策略决策请求事件的描述（或它们的组合）。PDP 对决定进行响应，PEP 通过接收或拒绝原始请求的方式来对其进行操作。PDP 还可以发送与准入控制无关的附加信息，或者用出错消息进行响应。

为了做出决定，PDP 可以依次从目录服务或其他的服务［最显著的是认证、授权和计费（Authentication，Authorization，and Accounting，AAA）服务］中请求附加信息。PDP 的另一个功能是出口与监控和计费目的相关的信息。

同时，RFC 2750 更新了 RSVP，将策略数据作为其协议中的有效载荷，由路由器和 PDP 处理，但对其他所有实体来说都是透明的。使用图 A.4 来解释与 RSVP 的交互。

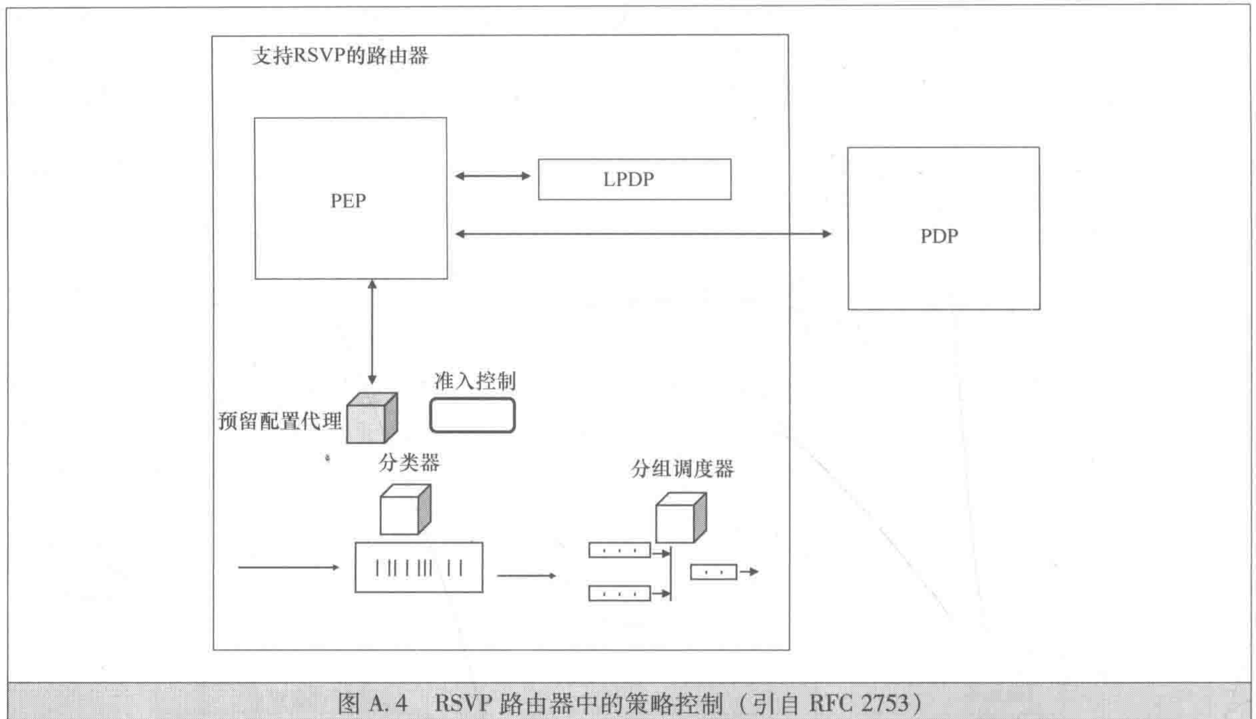


图 A.4　RSVP 路由器中的策略控制（引自 RFC 2753）

当与 RSVP 相关的事件需要做出策略决策时，预留配置代理会咨询 PEP 模块。后者首先检查 LPDP，以获得部分策略决策（如果可用），然后查询 PDP，附加上从 LPDP 获取的部分决策。最终的策略决策将返回给预留配置代理。

准入控制请求是一种典型的情况（通过它关联策略元素可以被传递)。然而，PDP 可以同时请求 PEP 向预留配置代理发出与策略问题相关的异常。例如，PDP 可以批准继续进行软状态安装的请求，并将预留向上游转发，但是会话时间可能受到限制，因此关于路径到期的相应通知也需要向上游传送。这个例子说明了 COPS 和 RSVP 之间重要交互的必要性。

因此，每当 PDP 返回错误时，它必须指定产生准入控制请求的事件是按照常规进行处理（但是添加了错误通知）还是停止处理。

相反地，当先前做出的决定需要改变或检测到错误时，PDP 本身可以自己发起与 PEP 的通信（通过发出通知)。如果通知需要沿着预留路径传播，则 PEP 必须将该信息传达给预留代理。

为策略的一般管理、配置和执行而创建的实际 COPS 协议与上述框架一致。

与 SNMP 或 CMIP 相比，COPS 中的新内容体现在，COPS 使用有状态的客户端 - 服务器模式，这与



远程过程调用的模型不同。与所有的客户端 - 服务器模型一样，PEP（客户端）向远程 PDP（服务器）发送请求，PDP 以决定予以响应。但所有客户端 PEP 的请求都由远程 PDP 安装和记住，直到被 PEP 明确删除。这些决定可以以单个请求的一系列通知的形式出现。实际上，这引入了一个新的行为：两个相同的请求可能会导致不同的响应，因为系统的状态在第一个请求和第二个请求到达时可能不同，具体取决于已安装的状态。COPS 的另一个有状态的特征是，PDP 可以将配置信息“推送”给客户端，然后将其删除。

COPS 状态模型支持两种策略控制机制，分别称为外包模式和配置模式。通过外包模式，PEP 在每次需要决策时查询 PDP；当采用配置机制时，PDP 规定了 PEP 内的策略决策。

与 SNMP 不同，COPS 旨在利用自识别对象，因此具有可扩展性。COPS 也可以在 TCP 上运行，确保重新传输。虽然 COPS 可以依靠 TLS，但它也有自己安全的机制，用于认证、防止重放攻击和消息完整性等安全操作。

人们发现 COPS 模型在电信领域非常有用，其中应用了该模型并将其进一步扩展实现了对 QoS 的支持。就云计算而言，COPS 的主要应用是 SDN。

A.1.3 网络配置模型和协议 ★★★

图 A.5 展示了 NETCONF 架构。

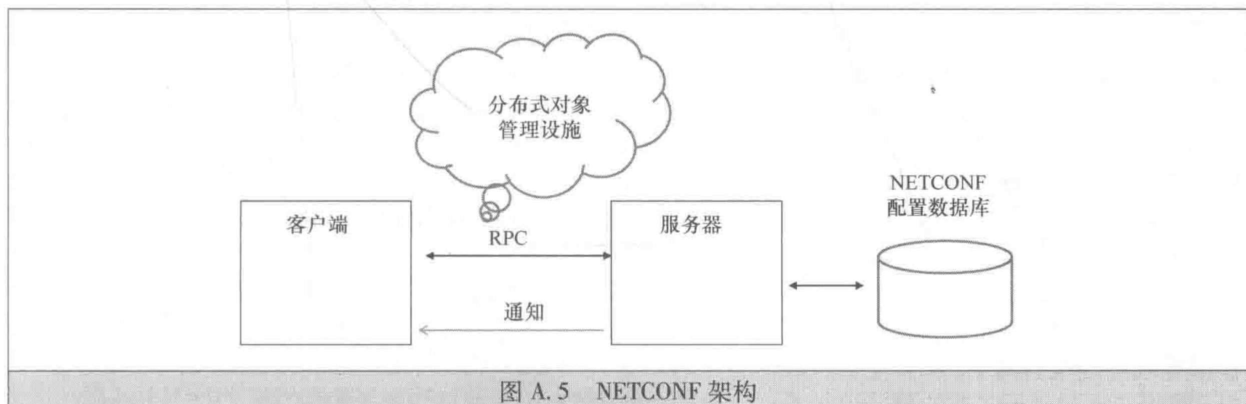


图 A.5 NETCONF 架构

至少在开发之初，该架构使用的是命令行界面，这解释了它所特有的一些特征。该协议遵循客户端 - 服务器模式，因为客户端发出命令，该命令在远程过程调用中被携带，并且从返回中获取结果。服务器通过配置数据存储区执行命令。RFC 6241 将此术语定义如下：“数据存储区持有将设备从初始默认状态转变为目标操作状态所需的完整配置数据集”。反过来，数据存储区又被定义为“存储和访问信息的概念上的位置”。就实现而言，“数据存储区可以实现，例如，使用文件、数据库、闪存位置或它们的组合”[㊀]。

在这种设计中，客户端 - 服务器协议和 CLI 都有两个主要的出发点。NETCONF 架构背离了严格的客户端 - 服务器模式，该架构支持异步通信，它从服务器上获取通知。这些是客户端需要关注的有效中断，因此客户端必须被设计以提供适当的终端处理程序。NETCONF 也背离了 CLI 模式，NETCONF 使用了结构化的 XML 编码数据。总之，NETCONF 规定[㊁]了一种面向对象的分布式基础架构。

这种基础架构以及所有其他的 NETCONF 部分可以很方便地展现在四层结构中，如图 A.6 所示。遵循 RFC 6241，各层由相应的示例进行说明，如图中右侧所示。

在该分层结构中，底层被称为安全传输层（需要注意的是，所有这 4 层都是 OSI 应用层中的模块。将“传输”[㊂]一词作为这层的名称是不幸的，因为该名称与 OSI 层次结构中所用的术语相冲突。但是，如果人们记住 NETCONF 从严格意义上来说是应用层协议，并且“安全传输”这一关键词的重点是

㊀ 不是很清楚为什么这个定义需要将操作系统对象（文件）、具体的文件系统（数据库）和物理设备（闪存）混合在一起，但是能够理解其中的要点。

㊁ 图中的云象征着这种基础设施，所以这个符号在这种情况下与云计算无关。

㊂ 特别有趣的是，这里“传输”的实际含义是“会话”。但是，实际上，在整个 RFC 以及其配套的 RFC 中都使用了“会话”一词。这意味着会话一旦建立，在会话的环境下会发生很多其他的事情（例如，订阅通知）。

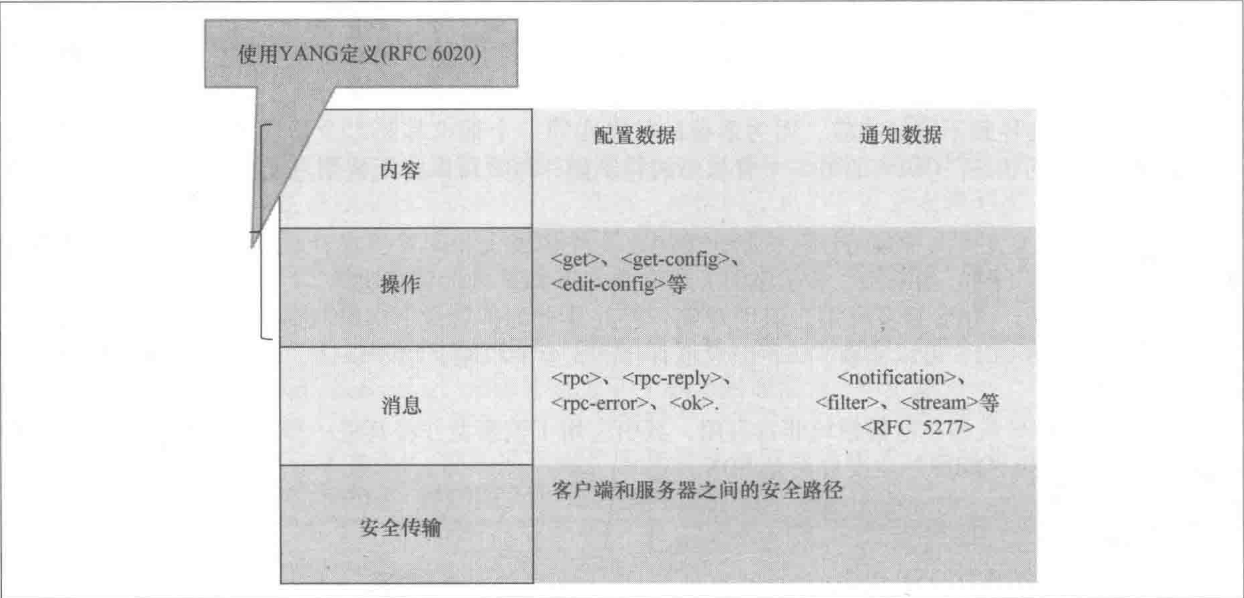


图 A.6 NETCONF 分层

“安全”，那么就可以避免混淆）。这个术语真正体现的是，所有的 NETCONF 消息都必须通过安全接口进行传递。最初，这个例子是由安全外壳（Secure Shell，SSH）协议支持的，而不是 Telnet，其中 CLI 命令以明文的形式自由地进行传输（再次强调，SSH 是为 CLI 开发的。通过 SSH，人们可以访问 Shell 解释器，否则只能通过不安全的 Telnet 来进行访问）。

然而，随着时间的推移，其他协议（特别是 TLS）已经被人们所接受。该层必须提供认证、数据完整性、机密性和重放防护功能。

下一层——消息，它只是一种传输无关的框架机制，用于对 RPC 相关的和通知相关的结构进行编码。图 A.6 示例条目中列出了 4 个 RPC 结构中的要素。通知要素列在 RFC 5277 中。接下来，将给出一个 RPC 调用规范的例子。

图 A.7 同时展示了客户端 deck - the - halls 方法调用和服务器回复的 XML 编码，其中前者中的单个参数是字符串 boughs_ of_ holly。首先，< rpc >元素带有一个强制属性 message - id，它是一个字符串，目的是唯一标识这条消息（以便使用相应的 < rpc - reply > 回应）。该字符串由 RPC 的发送方选择。本例中将其选为 123，遵循整数编码的传统。

< rpc >元素由 NETCONF 定义，相应的命名空间由 xmlns 字符串引用。该方法的名称 deck - the - halls 遵循这一规则。由于引入了这种方法，所以必须在 xmlns 字符串的后面提供命名空间的 URI。在该例中，唯一的参数（调用了 < fixture >元素）是字符串 boughs_ of_ holly，一个常量值。

```
a) <rpc message-id="123"
       xmlns="urn:ietf:params:xml:ns:netconf:base:1.0">

       <deck-the-halls
        xmlns="http://example.net/Cloud/deck-the-halls/1.0">
           <fixture boughs_of_holly>
           </fixture>
       </deck-the-halls>
   </rpc>

b) <rpc-reply message-id="123"
      xmlns="urn:ietf:params:xml:ns:netconf:base:1.0">
      <ok/>
   </rpc-reply>
```

图 A.7 a) deck - the - halls RPC 方法调用与 b) 使用肯定的结果回复

在执行成功的情况下，使用<ok>元素返回<rpc - reply>（否则，使用<rpc - error>返回，在这种情况下，将在其中指定错误原因）。

回到图 A.6 描述的层中。在第三层——操作，NETCONF 定义了基本的协议操作。这些操作包括<get - config>、<edit - config>、<copy - config>、<delete - config>、<lock>、<unlock>、<close - session>和<kill - session>。其中，前四个不言而喻；每个名称中的动词指定了在配置（或其一部分）上执行的操作。在回复参数的选择中有很多细微差别，RFC 为此给出了很好的解释。

<lock>操作是要求服务器拒绝（可能在较短时间内）来自其他客户端（包括 SNMP 客户端、执行 CLI 脚本的客户端或人类用户）对整个数据存储区修改的请求。在<unlock>操作发出之前或者取消会话持续时间内永久锁定的系统之前，锁定（lock）是有效的。如果数据存储区被锁定，那么来自其他客户端的<edit - config>、<copy - config>或<delete - config>请求将被拒绝（服务器不负责信号或监视设施的维护，这是客户端的工作）。

然而，在服务器端有一种更微妙<lock>操作执行方法，稍后将介绍这个内容。对于其余两种操作，<close - session>用于释放所有锁定和其他资源，并终止底层的安全传输会话（这似乎有点尴尬，因为没有<open - session>操作与之对应[㊀]；这里的总体想法是强制释放锁定和其他的资源）。<kill - session>实现的效果与之相同，但是在与其他的一些会话中，只有发出这个操作的客户端拥有适当的授权，这一操作的执行才可能会成功。

上一段提到的微妙指的是，在配置更新方面，NETCONF 支持并部分实现了称为 ACID［原子性（Atomicity）、一致性（Consistency）、隔离性（Isolation）、持久性（Durability）］的事务模型（有关此问题的详细内容，请参见本附录参考文献［1］的第 20 章）。简而言之，这些属性指的是在分布式环境中构成事务的一组操作。顺便说一下，这是一种经典的业务流程架构。

这里第一个主要问题是确保整个操作成功——考虑当一个或多个主机在事务中突然崩溃时，在分布式环境中可能发生的情况，或者在出现故障的情况下，所有已经执行操作的效果都可以被撤销。

典型的例子是从取款机中提取现金。这涉及借记账户和分配现金。如果机器无法发出现金，则不得将现金从账户中扣除，即借记账户；反过来，机器在借记账户（就是扣除现金）之前，不得退出现金。

原子性是解决这个问题的属性。它是通过在实际执行这些更改之前，记录和跟踪所有成功事务操作的效果并将其作为候选项来实现的。当有一个操作失败时，日志刷新，因此事务可以回滚到初始状态。否则，当所有操作成功完成并且将所涉及的操作项目提交给事务时，它才得以完成（即时服务器在最后一个阶段崩溃，重新启动后，它仍然可以根据日志，执行提交的事务）。

这里的一致性属性意味着，如果在事务开始之前，关联的数据一致，那么这些数据在事务完成后也将保持一致。隔离属性用于处理未完成事务中步骤的可见性，它是依赖于实现的。持久性涉及的是处理崩溃系统状态的保存（例如，在日志中）。

为此，NETCONF 将其<commit>操作的效果定义如下：

“<commit>操作指示设备实现候选配置中包含的配置数据。如果设备无法提交候选配置数据存储区中的所有更改，则运行配置必须保持不变。如果设备提交成功，则必须使用候选配置的内容更新运行配置”。

NETCONF 定义了<rollback - on - error>功能，在这种情况下，<edit - config>操作的<error - option>参数可以被设置成<rollback - on - error>。为了避免在共享配置的情况下不一致，RFC 建议客户端锁定配置。这就是标准明确禁止授权锁定的原因，“如果满足以下条件之一：

1）任意一个 NETCONF 会话或另一个实体已经拥有了一个锁定；

2）目标配置是<candidate>，它已被修改，并且这些更改尚未提交或回滚；

3）目标配置是<running>，另一个 NETCONF 会话有一个正在进行确认的提交。”[㊁]

正如看到的，操作层是相当重要的（查阅 RFC 6241 的所有 112 页内容可以明确证实这一点，其中更多的是相应的编程工作）。对于处理这种实现的开发人员来说，需要一种程序化的表示，这就是构成

㊀ 会话的开启是通过建立安全传输连接和发现服务器支持的 NETCONF 功能（一组指定的协议扩展）来实现的。

㊁ <confirmed - commit>是另一个 NETCONF 定义的功能。

该模型结构的第四层——内容。但是 RFC 6241 没有定义这一层，而是将这项工作指定给另一个标准化工作，由其指定“NETCONF 数据模型和协议操作，涵盖操作层和内容层”。幸运的是，IETF NETMOD 工作组已经完成了这样的一个规范，称为 YANG㊀，发布在 RFC 6020 中。

在本书中，将无法对 YANG 的内容进行详细的论述，但是感兴趣的读者会发现 RFC 6020 的内容写得非常不错。我们只注意到，YANG 实际上是 NETCONF 的建模语言。它使用了很好的结构，所以用其进行建模之后，人们可以发现它的高级视图和 NETCONF 操作中的最终编码。通过设计，YANG 还可以进行扩展，从而允许其他 SDO 对其扩展进行开发，并且让单个编程人员来生成即插即用模块。

YANG 还保持（有限地）与 SNMP 的兼容性：SMIv2 MIB 模块可以自动地转换为 YANG 模块进行只读访问。

自然地，NETCONF 被广泛用在 SDN，用于配置虚拟交换机。如第 7 章所述，有一些 NETCONF 插件可用于这一目的，这些插件作为开源 SDN 项目 Open Daylight 的一部分。

A.2 TOSCA 业务流程

前面已经介绍过，云应用程序的拓扑结构和业务流程规范（Topology and Orchestration Specification for Cloud Applications，TOSCA）是一种 OASIS 标准[2]。在撰写本书（英文原版）时，它已被业界所采纳。TOSCA 是一种生命周期软件管理的 DSL，还是一种建模语言，它描述了服务的结构（一个模型），用于表达要运行它而不是以程序化的方式实现时需要完成的工作，至于如何实现则由解释器自动生成。

这些类型的语言没有一夜之间出现。人们对特定域建模更为普遍的看法是，特定域建模是一种通过自动代码生成来提高软件生产率的手段，这种源自高级规范的看法得到了特定域建模（Domain - Specific Modeling，DSM）论坛的支持，该论坛网站提供了很多有关这方面内容的有趣信息（包括权威性的参考书目）。

本书从历史的角度回顾了多个关键技术，在特定域语言的发展方面，最重大的突破是在 20 世纪 70 年代来实现的。在开始撰写本书（英文原版）时，与我们第一次见面的 Noah Prywes 教授曾经发表了一篇开创性的论文[3]，介绍了“模块描述语言（Module Description Language，MODEL），该语言旨在供管理、商业或会计专家使用，而不需要它们接受任何计算机方面的培训”。MODEL 描述了“输入、输出和与系统规范相关的各种公式”，但是没有提供编序信息。后者是 MODEL 处理器的工作，由它负责生成代码（可能在解决与编程人员交互过程中的不一致或歧义之后）。在 20 世纪 80 年代和 90 年代，宾夕法尼亚大学的 Prywes 教授和他的研究生设计了一个完整的分布式软件生命周期规范系统，该系统还可以用于逆向工程[4]。这是一个成功的技术转让例子，基于研究成果的实际 MODEL 产品，由计算机指挥和控制公司进行开发和销售。

回到 TOSCA。借助图 A.8 来解释 TOSCA 在应用程序业务流程中的作用，该图是基于 Sivan Barzilay 提供的宝贵资料做出的。预计这里的一个应用程序将使用多个虚拟机（包括虚拟网络设备），以特定的方式相互连接，并由一组策略进行排列。这里的典型示例是运营商网络中部署的虚拟网络功能（如 IMS）。

在底层“基础设施层”的管理网络功能虚拟化过于复杂，正如在第 7 章讨论电话网络管理演进时演示的那样。在 IaaS 层，我们知道如何编排堆栈的业务流程。具体来说，熟悉 OpenStack 的业务流程机制，这些机制当然适用于基于 OpenStack 软件云节点（数据中心）。但是，如果存在一些部署了其他实现的节点呢？为了在这种情况下保持统一的服务规范，我们需要在转换器层终止 Heat API，其中后端功

㊀ “阳（YANG）”是中国哲学中两个对立辩证且内在关联的概念“阴（YIN）和阳（YANG）”中的一个，这些概念起源甚早。早在公元前 700 年左右就出现了。秦国时期的《易经》中写道，“阴”代表的是“女性、被动、负面”，而“阳”则代表了“男性、主动、正面”。例如，太阳是阳，而月亮是阴。阴和阳是不可分割的。类似地，RFC 6020 将“YIN（阴）”定义为“YANG（阳）”（一种数据建模语言）的一种基于 XML 的表示形式：“YANG 模块可以被转换为一种等效的 XML 语法，称为 YIN（YANG Independent Notation，YANG 独立标记法）（摘自 RFC 6020 中第 11 节），允许使用 XML 解析器和可扩展样式表语言转换（Extensible Stylesheet Language Transformations，XSLT）脚本的应用程序对模型进行操作。这种从‘YANG’到‘YIN’的转换是无损的，所以 YIN 的内容可以回溯到 YANG”。

能将接管转换任务。

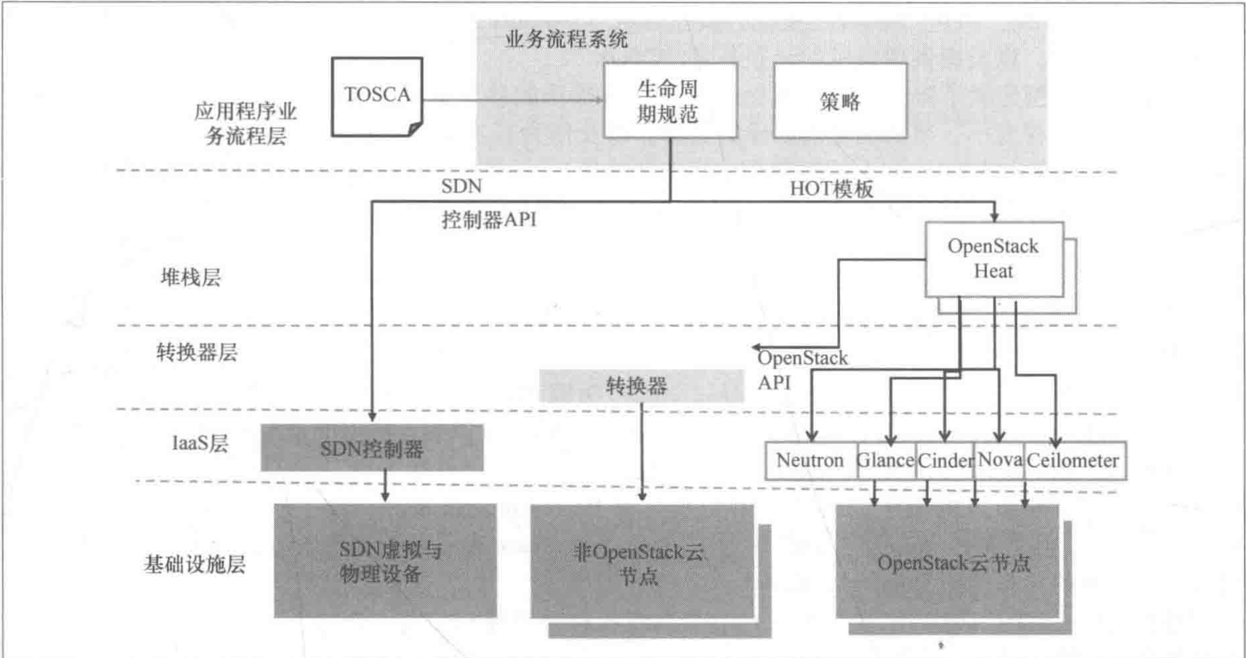


图 A.8 业务流程分层框架（由 Sivan Barzilay 提供）

在堆栈层，将虚拟化与数据通信（和 SDN）相结合。因此，需要将堆栈的业务流程与网络拓扑结构实体进行整合。再次强调一下，这一方面对于网络功能虚拟化至关重要，其中应用实际上是网络即服务。

自然地，为了将广域网与堆栈业务流程进行整合，需要另一层抽象和规范，以便在不同的平台上实现对这些服务的移植。这正是 TOSCA 旨在实现的目标。

核心的 TOSCA 规范[2]描述了服务组件以及这些组件之间的关系。另外，也是关键的一点——规范语言允许通过管理程序为创建和修改业务流程中的服务指定操作行为。因此，TOSCA 服务模板描述了在整个生命周期内跨不同环境保存的拓扑结构和业务流程中的恒定量。

要想了解更多细节，可以先从命名空间开始。TOSCA 使用 XML 并定义了两个命名空间前缀：默认的前缀是 tosca；另一个前缀是 xs。TOSCA 扩展机制支持从其他命名空间导出实体（属性和元素），因为它们不能与 TOSCA 命名空间的任何实体相冲突。

图 A.9 对服务模板的抽象语法结构进行了描述（后面将通过一个具体的例子，来对抽象的含义进行说明）。

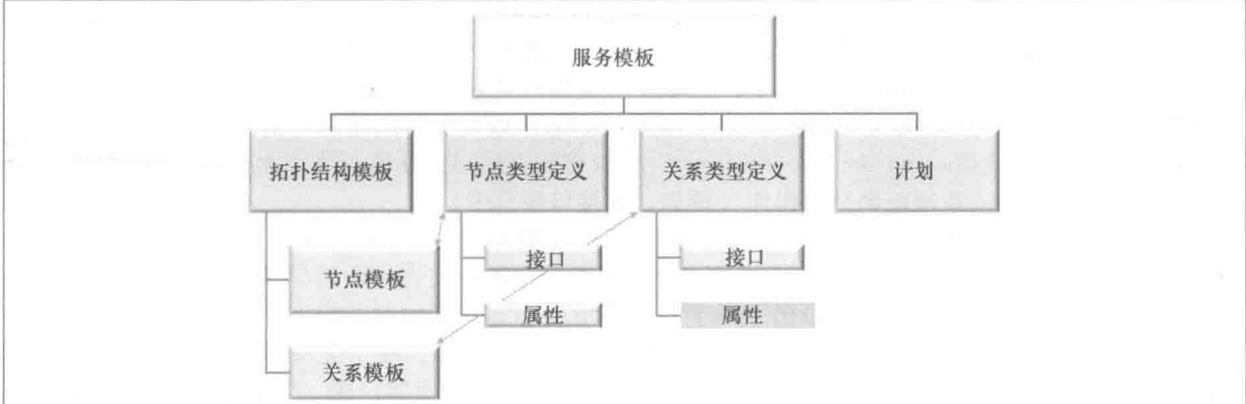


图 A.9 TOSCA 模板结构

服务的基本组件称为节点，节点类型（以及关系类型、拓扑结构模板和计划）在最高级别的规范

中被声明。例如，部署在云上的 Web 服务中的节点类型可以是名为 My_ App 的 Web 服务器应用程序；一个 Web 服务器 X_ Web_ Server；底层的操作系统 Y_ Linux（Linux 版本）；承载应用程序的虚拟机 Virtual_ Machine；以及提供虚拟机的云服务 Z_ Cloud。

每个节点类型定义了服务组件的属性以及要执行操作的各个接口。在服务器型节点中，属性包括 CPU 的数量、内存大小、要实例化的映像的名称，以及作为基本安全属性的 SSH 密钥对的位置（参见图 A. 11）（可以通过指定的输入过程获取上述参数的值）。这些接口指定了服务生命周期内在节点上执行的操作。每个操作（例如，创建、开启或停止）与指向实际实现该操作的脚本的指针一起出现，例如：

```
create: scripts/server_library/install_server.sh.
```

这是 TOSCA 中提供程序插件接口的地方。关系类型规定了给定类型节点之间的关系（或连接），其思想是服务是一个有向图，图的顶点是节点，边是关系。

关系类型规定了哪些节点可以连接。通过明确声明源和目标元素来表示方向（稍后将看到，方向对于建立处理顺序至关重要）。接口部分允许我们插入代码，就像节点类型的情况一样。

这种基于图的表示（以及通过遍历图得到的后续动作推导）是在 TOSCA 中使用“拓扑结构”一词的原因。使用早期节点的例子，可以使用关系类型 HostedOn，如 My_ App [is] HostedOn X_ Web_ Server 中所示。但是 My_ App 也可以使用由节点 My_ Database 表示的数据库服务，为了指定这种关系，将引入一种新的关系类型 QueriesDatabase[⊖]。

总体来说，TOSCA 拓扑结构模板由一组节点模板和关系模板组成。图 A. 10 对 Web 服务示意图的部分拓扑结构模板进行了说明。

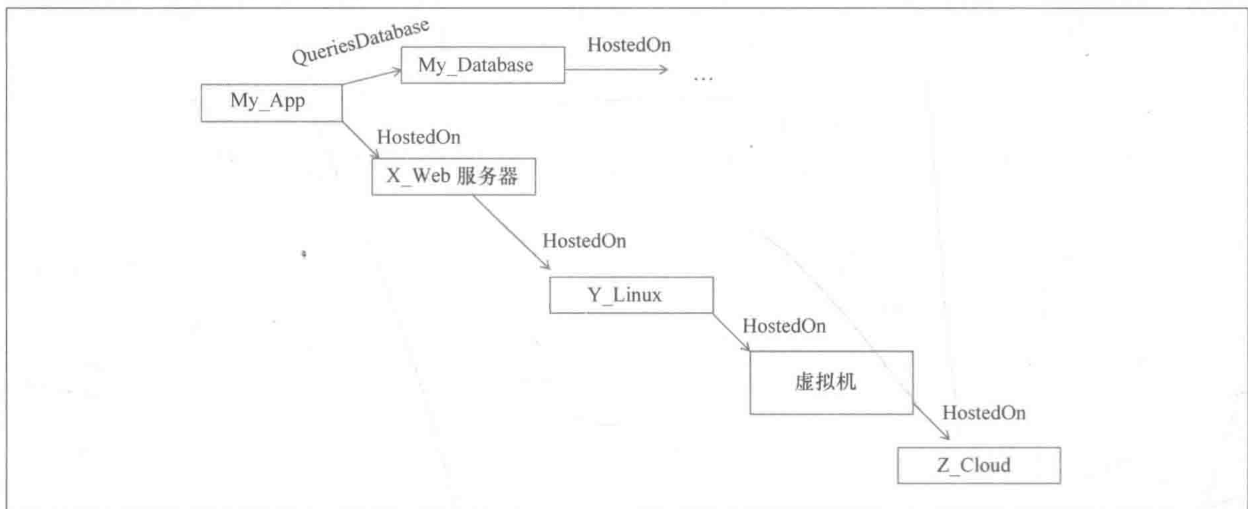


图 A. 10　拓扑结构模板示例

TOSCA 服务模板的最后一个要素是计划。这里通过工作流的概念对服务实例的管理方面的内容进行了定义。在选择一种规范语言的情况下，TOSCA 可遵循其他标准，例如 OASIS 自己的 Web 服务业务流程执行语言（Business Process Execution Language，BPEL）或对象管理组制定的业务流程建模标记法（Business Process Modeling and Notation，BPMN），尽管它可以使用其他的语言。无论哪种方式，工作流都是指定义为节点和关系模板部分的操作（例如，在接口规范中）。

图 A. 11（由 Sivan Barzilay 提供）是将 TOSCA 模板转换为可以由 AWS CloudFormation 和 OpenStack 理解的 HOT 模板的简单示例。但是，怎样将一个规范解释成能够支持生命周期操作顺序的方法？节点模板对如何部署和管理节点的问题给出了一个相当简单的答案。这种关系规定了处理节点模板的顺序（并且，正如前面提到的，它可以支持注入额外的处理逻辑）。例如，对于类型为 HostedOn 的关系，主机自然应该在该主机承载的节点之前进行创建和配置。类似地，对于客户端 - 服务器关系，服务器必须在其所有客户端之前进行处理。

⊖　这种关系类型通常被命名为 ConnectsToDatabase。

a)

```
tosca_definitions_version: tosca_simple_yaml_1_0_0_build_1

description: My most simple server template

inputs:
 image_name:
   type: string
   default: tinyCentos
   keypair_name:
   type: string
   default: TEST-KEYPAIR-FOR-BDD

node_templates:
   server1:
   type: Ournodes.Compute
   properties:
   num_cpus: 1
   mem_size: 1024
   image_name: { get_input: image_name}
   keypair_name: { get_input: keypair_name}
```

b)

```
heat_template_version: '2013-05-23'
resources:
 server1_119677379087827598:
  metadata: {deploymentId: '119677379087827597',
      deploymentNodeId: '119677379087827598'}
  properties:
   availability_zone: zone0
   flavor: default
   image: 41d1769d-a6fc-4849-8c70-8f5b0d64fcb5
   key_name: TEST-KEYPAIR-FOR-BDD
   metadata: {deploymentId: '119677379087827597',
      deploymentNodeId: '119677379087827598'}
   name: cPaaS_server1_119677379087827598
   networks:
   - port: {get_resource:
      server1_119677379087827598_mgnt_port}
  type: OS::Nova::Server
 server1_119677379087827598_mgnt_port:
  properties:
   fixed_ips:
   - {ip_address: 10.38.237.63, subnet_id: 66bc11c4-2312-4c55-
      a169-20ca0d21f5f1}
   network_id: 8796324e-7a75-43f7-8be8-32581d846f5c
   security_groups: [d8be0483-77da-41b5-bc17-a55c2eaa0384]
  type: OS::Neutron::Port
```

图 A. 11 a）TOSCA 模板转换成 b）相应的 HOT 模板示例（由 Sivan Barzilay 提供）

有关这方面的更多内容，推荐读者参阅一篇不错的论文[5]，该论文还从 TOSCA 模板的角度介绍了服务的生命周期，该模板是在服务提供阶段由云服务提供商创建的。

为了采用 TOSCA，业界已经做出了巨大的努力。在这方面有一个开源项目，被称为 OpenTOSCA。从作者的角度来看，OpenTOSCA 生态体系的命名规则似乎源自于酿酒：除了 TOSCA 的运行时环境“OpenTOSCA 容器”，它还提供了一个名为 Winery 的图形建模工具和一个名为 Vinothek 的容器中可用应用程序的自助服务门户。

正如前面提到的，人们在实现 OpenStack 与 TOSCA 的交互中完成了相当多的工作。TOSCA 还是一个正在研究的课题。其中一个项目是定义一套策略框架，将其作为一个重要的示例，用于指定安全策略。云服务认证是其中的一个主要的驱动因素（例如，本附录参考文献［6］中解决的问题）。一旦认证，服务即保持不变。因此，需要在正式的服务描述中得到认证要求。

需要注意的是，“TOSCA 缺少有关如何应用、设计和实现策略方面的详细描述”㊀，本附录参考文献［7］演示了如何来定义安全策略。这篇论文考虑了两种方法。第一种方法是基于计划的，因此需要修改构建、管理和终止计划中的工作流，以支持标注的策略。第二种方法不涉及任何计划的修改；相反，需要修改相关的操作。

但是，在研究的进行中，OpenTOSCA 已经在生产中成功应用。如本附录参考文献［8］所述，使用 OpenTOSCA 和 OpenStack 的企业内容管理（Enterprise Content Management，ECM）系统的设计、规范和云部署由一个研究生在 MS 项目的课程中就可以实现。

A. 3 REST 架构风格

本附录参考文献［9］介绍了 REST 架构风格的概念，同时在 Roy Fielding 博士论文[10]中的第 5 章也对这一概念进行了阐述。Fielding 博士致力于 HTTP 的设计、规范和实现，尽管 Fielding 博士在其合作制定的 HTTP 规范中已经强调了“REST 是一种风格，完全是协议无关的。默认情况下也不是基于 HTTP 的 API REST”，但是行业内仍固执地认为 REST 是“基于 HTTP 的”。

正如本附录参考文献［9］指出的，“REST 是一组架构约束，目标是最小化延迟和网络通信，同时

㊀ 当然，目前 TOSCA 标准的版本确实可以让我们制定策略，但是如本附录参考文献［7］所述，这些规范仅仅是注释，策略定义和处理的各个方面都可以解释。

最大化组件实现的独立性和可扩展性。REST 实现了交互的缓存和复用、组件的动态可扩展性，以及中间件对动作的处理，从而满足了互联网规模分布式超媒体系统的需求”。

这里的“缓存”和“中间件”是万维网中的典型架构实体，为了讨论 REST，需要回顾万维网的体系结构。我们将会看到，理解这种风格的关键在于这种表述最后几个字：“分布式超媒体系统”。在下一节中将回顾后一个概念；后两小节简要介绍了万维网架构的各个方面，并概述了 REST 风格。

A.3.1 超媒体的起源与发展 ★★★

超媒体这个词在这里指的是音视频参考系统。在语法上，超媒体是超文本，因为实际链接是 ASCII 编码的。1945 年，Vannevar Bush 教授在其著名的文章[11]中就表述了早期版本的超媒体概念：

“可以想像出一个未来人人都会使用的设备，它是一种机械化的私人文件和库。需要给它取个名字，就叫 memex 吧。memex 是一种个人存储他所有的书籍、记录和通信的设备，通过机械化实现了惊人的处理速度和灵活性。这是对它内存的一个巨大的补充”。

设想的实现方法（以这种形式未实现）是使用微缩胶卷作为“各种书籍、图像、当前期刊和报纸”的存储介质。这些将根据在打字机键盘上输入的代码进行机械的索引和搜索，并投影到屏幕上。超文本的关键概念——关联链接是该愿景的一部分：“它直接提供了到关联索引的方法，其基本思想是任何条目可以随意选取，并且可以自动地引用其他条目”。

1965 年，Theodor Holm Nelson 在其论文[12]中开始从事基于计算机㊀实现 Bush 概念的工作。本附录参考文献［12］中引入的“超文本”术语，“意思是以一种复杂方式相互交叉的书写或图像材料的整体，不能方便地在之上呈现或表现出来”。同样的，“胶片、音频和视频”它们“可以以非线性系统的形式进行组织，例如格子，用于编辑的目的或用于显示不同的重点”。与“超文本”一起出现的是“超电影”这个词，表示一种“可浏览或可变化顺序的电影”，它是“唯一的一种超媒体，需要引起我们注意”。

当然，这篇论文的意义远比仅仅定义这些术语要深远得多，它提供了信息结构、文件结构，甚至用来表示这种文件格式的语言㊁。

与此同时，宾夕法尼亚大学的两位教授 Noah Prywes（算上这次，已经第三次提到他了）和 Harry J. Gray 曾经一直从事于多列表（Multi - List）系统的构建工作，并将其在 1959 年的论文[15]中进行了概述，本附录参考文献［16］对这一内容给出了进一步的介绍，人们预想它可以通过基于软件的关联存储器实现来支持库函数自动化，其中所提及的内存以链表结构进行组织。本附录参考文献［17］报道指出，计算机图形学领域的先驱者 Andries van Dam 在其宾夕法尼亚大学的博士研究中曾经使用过多列表。1966 年，他以博士论文“图像数据的数字处理研究”进行答辩，获得了他第二个计算机科学博士学位。

之后，Nelson 和 van Dam 与美国布朗大学的学生团队一起，联手在 IBM 360[18]上开发了超文本编辑系统（Hypertext Editing System，HES）。这个项目后来成为了一个更高级的项目，但是这里我们知道，超文本处理的主要要素已经存在。该项目涉及了多用户访问（不仅用于读取，还可用于修改文件）。计算机终端上显示的文本可以一支光笔（鼠标的前身）进行选择、标记和注释。在验证超文本思想的基础上，HES 表明线性编辑（顺便提一下，即使在 20 年后，它也仍然会存在）可以用全屏幕编辑与格式进行替换。

快速跳到 1980 年，Timothy Berners - Lee 先生在 CERN 开发了基于超文本的系统 ENQUIRE。到 1989 年，他使用了一种不同的设计，将超文本技术与互联网结合在一起，建立了第一个网站；最后到 1990 年，他发布了第一款 Web 浏览器，被称为 WorldWideWeb。

这种架构的超媒体组件称为资源（最初是由文件表示的一种文档，但是后来演变为由程序运行产生的结果）。之前提到的通用资源命名方案，可以让我们通过 DNS 查找的方式查询服务器上的资源。客户端设备上的浏览器能够解释使用超文本标记语言（Hypert - Text Markup Language，HTML）编写的网

㊀ 这篇论文的一个亮点是，它准确地预测了计算机将成为一种用户友好的且成本低廉的书写和研究工具。

㊁ 有趣的是，大约在同一时间，Vladimir Nabokov（他不是一个计算机科学家，而是一个畅销作家）出版了一部小说——《Pale Fire（苍白的火）》[13]，这部小说使用超文本书写，如本附录参考文献［14］所示。

页，并且获取相应的资源⊖。

资源被分配了 RFC 3986 中定义的统一资源标识符（Universal Resource Identifier，URI）。URI 可以是统一资源名称（Universal Resource Name，URN），该名称对资源进行了唯一标识，但没有指定资源的位置。或者，URI 也可以是统一资源定位符（Universal Resource Locator，URL），该定位符实际上指定了资源的位置。其中，这里的“或者”不是排他的：URI 可以是 URN 和 URL。

URI 只是一个标识符，它不必被用来指定一个可访问的资源。与 UR 引用相关联的操作由协议和相应的元数据定义。RFC 3986 强调，系统对资源执行的典型操作包括访问、更新、替换或查找属性，这些操作由使用 URI 的协议定义。

URI 语法的正式定义如下：

```
URI = <scheme> ":" <hier-part> ["?" <query>] ["#" <fragment>].
```

其中的 scheme 通常是相关的协议（如 HTTP、mailto 或 SIP），但它也可以仅仅是一个字符串 URN，这意味着 URI 是一个 URN。< hier - part > 包含资源所在主机的 DNS 名称、要使用的端口（这是可选的，默认为 80），以及资源的路径（仿效 UNIX 文件系统）。其余部分是可选的搜索部分。图 A.12 列举了几个 URI 的示例（需要注意的是，从前两个示例可以看出，单个资源可以拥有不同的 URI）。

- ftp://ftp.ietf.org/rfc/rfc3986
- http://www.ietf.org/rfc/rfc3986.txt
- ldap://[2001:db8::7]/c=GB?objectClass?one
- mailto:Cloud.Administrator@example.com
- news:comp.infosystems.www.servers.unix
- tel:+1-816-555-1212
- telnet://192.0.2.16:80/
- urn:oasis:names:specification:docbook:dtd:xml:4.1.2

图 A.12 URI 示例

这里还有一个例子说明了 URI 字符串中的搜索部分的用途。当使用 Google 搜索 an example of URL 时，浏览器会显示下面的 URI，结果如下：

https://www.google.com/webhp?sourceid=chrome-instant&ion=1&espv=2&ie=UTF-8#q=an%20example%20of%20url.

这个例子不仅突出了网络超媒体实现的灵活性，而且还演示了 URI 是如何存储和“驱动”应用程序状态的。

后者是 REST 架构风格的主要宗旨，稍后将继续介绍这方面的内容。从本附录参考文献［9］中可以看到：“超媒体因为它的简单性和通用性，被选作用户界面：无论什么样的信息源，都可以使用相同的界面，超媒体关系（链接）的灵活性允许无限的结构化，并且对链接的直接操作允许信息内的复杂关系通过应用程序来引导读者。由于大型数据库中的信息通常通过搜索界面而不是浏览器更容易进行访问，因此 Web 还通过向服务提供用户输入数据并将结果以超媒体的形式进行呈现的方法，来纳入执行简单查询的能力”。

规定网页结构使用超文本标记语言（HTML），在这种网页结构中还提供了相关资源的表示。W3C 完成了 HTML 的标准化工作，W3C 是由 Berners - Lee 先生成立和领导的组织。最新的版本 HTML 5.0 可以支持本地（而不是基于插件的）音频和视频、浏览器存储和在线矢量图形。

元数据是资源表示的一部分（例如，媒体类型或最后修改的时间）。还有一种数据组件是控制数据，下一节回顾并介绍数据缓存时，控制数据的目的将变得清晰明白。

A.3.2 万维网架构要点 ★★★

从一开始，Web 设计就考虑了降低带宽的使用率（这也意味着更快地访问资源，并且在很多情况下，它还意味着减少网络费用）。在分布式环境中，这可以通过一种有效的方法，即在一个或多个高速缓存中复制数据来实现。数据的缓存从客户端开始，网页一旦被获取，就可以被浏览器存储起来，进一步的复制由代理执行。

图 A.13 举例说明了企业、网络提供商、服务提供商部署的代理情况（应当注意到，部署的代理不仅可以为缓存提供物理存储，还可以实现内容过滤、协议转换、收集分析和终端用户匿名性等今天在

⊖ 当然，客户端不一定是浏览器，它是通向人类用户的网关。而用户可以是一个自动机（进程或线程），它是调用相关 API 的地方。但是，现代浏览器（如 Chrome）一般都支持基于 API 的编程接口。

万维网使用中的所有重要功能。毫无疑问，伴随着这些功能一并出现的还有一些众所周知的缺点，即这些功能被滥用导致的问题）。

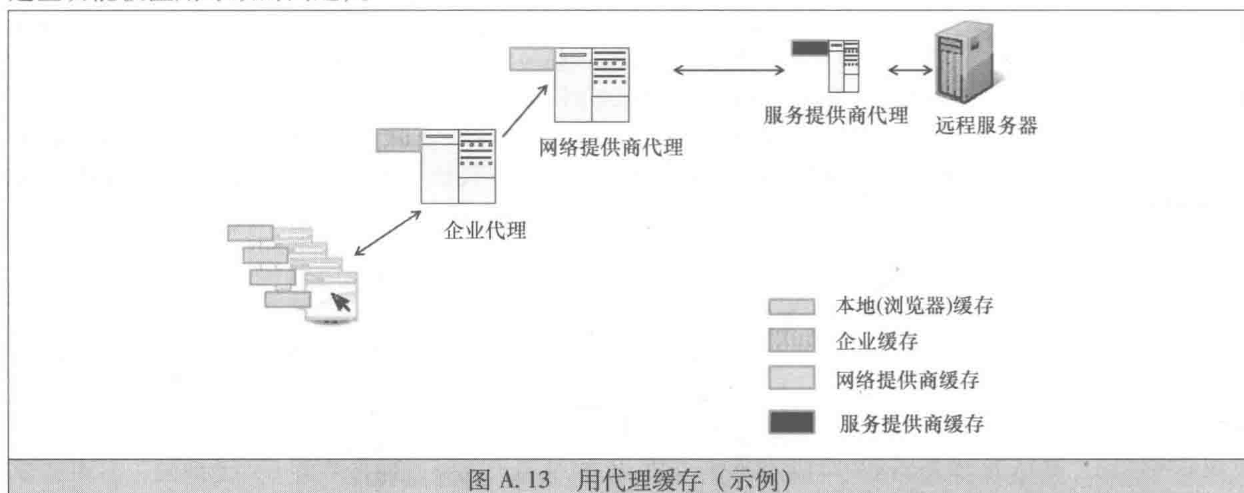


图 A.13　用代理缓存（示例）

在极端情况下，当服务器关闭时，缓存仍然可以提供其服务，从终端用户的角度来看，这种情况类似于观察一颗变暗的明星发出的光。

由于使用缓存的端点（客户端或代理）必须知道缓存是否有效（即包含与服务器所含的相同的信息），所以缓存的存在必然限制了部署在这种架构中的协议使用。

另一个限制源于网页的动态性质。最终的 HTML 文档不必存储在服务器上，它可以在服务器端，也可以在客户端，或者在两者之上同时动态地创建，其中客户端和服务器端分别创建其各自的部分。

为了说明万维网的工作原理，有必要强调它最原始的协议，即 RFC 2616 中定义的 HTTP。HTTP 运行在 TCP 上（目前使用持久连接），因此它具有“可靠的管道”[㊀]。HTTP 是一种请求/响应协议，其请求由客户端发出，服务器返回响应。

所有的 HTTP 消息均经过 ASCII 编码。消息首部实际上是纯 ASCII 文本，而消息体可以包含 ASCII 编码的二进制数据。

HTTP 一直致力于实现面向对象的范式。因此，它定义了一组方法列表[㊁]，这些方法作用于请求 URI 标识的资源：

1）GET，获取资源表示。

2）HEAD，只获取 HTTP 首部（通常用于检查与资源相关联的元数据）。

3）POST，请求源服务器接受将封装在请求中的实体作为新的资源附属（稍后将介绍一个相关的解释示例）。

4）PATCH[㊂]，部分修改资源的表示。

5）DELETE，请求删除资源。

6）OPTIONS，请求与资源相关联的请求/响应链上可用的通信选项相关信息。

7）TRACE，启动原始消息的回送。

8）CONNECT，请求连接到可以作为隧道的代理。

使用几组预留的代码对响应结果进行分组：

1）100～199：信息。

2）200～299：成功并携带数据（如果存在数据）。

3）300～399：重定向。

㊀ 这样的管道无法提供保密、非重放或完整性的服务。这样的服务以及服务器认证可以通过在 RFC 5246 中标准化的传输层安全（Transport Layer Security，TLS）协议之上运行 HTTP 来实现。TLS 运行在 TCP 之上，并在客户端和服务器端之间提供一种安全通道。本身，HTTP 就可支持客户端认证的机制。

㊁ 允许新方法的扩展名。

㊂ 这种方法，即 2010 年完成的一种扩展标准化，发布在 RFC 5789 中。



4）400 ~ 499：客户端报错（还可用于请求认证，如401，伴随着对客户端身份的质疑）。

5）500 ~ 599：服务器报错。

这些响应码大部分的含义都比较简单，但其中有两类相对比较复杂。第一个，重定向是一个强大（并且也是存在潜在安全隐患）的功能，顾名思义，该功能用于指示客户端访问另一个服务以获取信息。其中的处理可能很复杂，例如，必须避免死循环，在某些情况下，浏览器本身无法决定是否允许重定向。然而，重定向可以作为构建服务的工具。将在本章最后一节中看到，基于HTTP的API系统地使用了重定向来实现基于令牌的身份认证管理方案，例如，开放授权（Open Authorization，OAuth）协议中使用的身份认证管理方案。

第二个，客户端报错，不能用来指示错误，而是要求额外的处理。例如，401响应用来请求对客户端的认证，向客户端提供了一个质疑，由客户端进行回答以证明自己的身份。类似地，从代理的角度来看，407响应调用了相同的过程，除了客户端必须面向代理进行身份认证。

前面提到的CONNECT暗示了HTTP可以感知到代理的存在，有关这一方面还有更多的内容。缓存处理的细微差别体现在首部中。关键信息元素（通常）包含给定页面的哈希㊀值，这样的元素被称为实体标签（Entity Tag，ETag），因为它被用作页面内容的标签。ETag可用于检查缓存的新鲜度。当首次（通过GET方法）获取资源表示结果时，其ETag将与缓存的表示结果一起存储。随后的GET请求携带带有if - none - match的请求首部字段，设置请求条件：如果页面是新的（即它的哈希值与ETag中包含的哈希值相同），则不需要再传送这个页面。通过PUT使用相同的机制，可以防止客户端不了解的页面被篡改。

保持时间值也有助于验证新鲜度。返回响应中携带的最后修改的实体首部字段记录了页面被修改的日期和时间。客户端可以在if - modified - since首部中使用具有相同值的条件GET。启用条件处理的数据元素在本附录参考文献［9］中被称为控制数据。

A.3.3 REST原理 ★★★

这些原理适用于在Web架构存在的情况下对服务进行编程的做法。首先，不可能指定浏览器实现，因此客户端关注的问题必须和服务器关注的问题分开——不应该由客户端到服务器的接口绑定。这与RPC方法截然不同，在RPC中这样的绑定是被明确指定的。

考虑到Web服务的规模，服务器无法为每个客户端保持应用程序的状态，服务器必须是无状态的（另一个原因是，服务器可能被复制，通过服务器实例实现负载均衡）。因此，从客户端到服务器的每个请求都应该包含服务器理解请求所需的所有信息。

对于REST来说，不是每个提供状态相关信息的机制都可以接受。HTTP中有一个众所周知的机制，称为cookie，在RFC 6265中有相应的规定。cookie是一种描述客户端状态的数据结构，服务器放置一个cookie用来对客户端的初始请求进行响应，之后将在所有后续的交互中交换（并可能被服务器更新）这一cookie，以便cookie保持客户端的状态。这种方法不符合REST风格，它规定了通过超媒体（URI）驱动应用程序状态转换的方法。实际上，本附录参考文献［10］在其第6.3.4.2节中对此给出了一个鲜明的例外，注意到与用户备份的冲突（如在浏览器中单击“后退”按钮），“备份到cookie之前的视图，浏览器的应用程序状态不再与cookie中表示的存储状态相匹配。因此，发送到同一个服务器的下一个请求将包含一个会误导当前应用程序上下文的cookie，导致双方发生混淆”。

在有中间件和缓存的情况下，应当取消额外的交互，以便最贴近客户端的正确缓存可以响应并消除进一步的请求传播。为了实现这一点，来自服务器响应中的所有数据，应当被标记为可缓存的或不可缓存的。

客户端与服务器之间的接口受到以下限制：资源的识别，通过其各自的表示对资源进行处理操作，自我描述消息㊁，以及将超媒体作为状态转换的手段㊂。

REST风格还规定了分层结构（封装传统服务，并保护来自传统客户端的新服务㊃）和按需代码的使用，即代码可以通过超媒体方式被带到客户端上执行。

㊀ 为散列，也可称为哈希，本书通用“哈希”一词。

㊁ 换句话说，这些消息可以被所有中间件（代理）所理解。

㊂ 本附录参考文献［10］（第5.1.5节）描述了这种约束：“超媒体作为应用程序状态的引擎”。

㊃ 例如，对于这种方案，使用基于Web的电子邮件服务，SMTP邮件服务协议被封装在Web用户不可见的层中。

REST 风格规定了固定资源名称或层次结构的定义（事实上，任何“类型化归类”的资源其中的类型都需要被客户端所理解）；构建恰当 URI 的指令必须来自于服务器，就像 HTML 表单或 URI 模板一样。实际上，这是为了禁止对任何具体协议的依赖。

最后一点经常被误解。在 Fielding 的博客中，他强调 REST API 必须以协议无关的方式定义。实际上，HTTP 不是唯一的 Web 应用协议。如前所述，IETF 认识到了单一事务请求/响应协议存在的不足［它不能支持诸如通知的异步“响应”，因此需要昂贵（在计算资源占用方面）且尴尬的轮询机制］，因此制定并标准化了一种称为 WebSocket 的新型全双工协议。

通过一个例子来结束这一节的内容，这个例子演示了超媒体是如何驱动服务状态的。为了让这个例子变得更为具体，明确使用 HTTP，同时演示 REDIRECT 方法的功能。

这里考虑一个简单但相当典型的服务，即用户订购一些可以获得回执的商品。用户请求由 URI X 表示的商品，得到一个表单作为响应，填写表单并提交。

如果该服务是按照图 A. 14a 来实现的，其中通过 GET X 获取表单并通过 POST X 将表单返回给服务器，此时应用程序将处于暂态状态。而这时浏览器可能会转到其他页面的显示上，这样用户将永远无法通过备份返回到之前页面的接收状态。不能确定表单提交是否成功，或者无法获取表单提交回执——用户可能会再次执行相同的步骤，从而再提交一次表单（所以，用户可能会支付两次，得到两个相同的商品，而用户只需要一个）。

图 A. 14b 描述了另一种实现方法，该实现方法解决了这个问题。这里 POST 将后缀 y 附加到 X 上，从而创建了一个附属资源 X/y。响应将用户重定向到 X/y，这表示一种新的持久状态，因此所有后续的 GET X/y 都将返回这一回执。

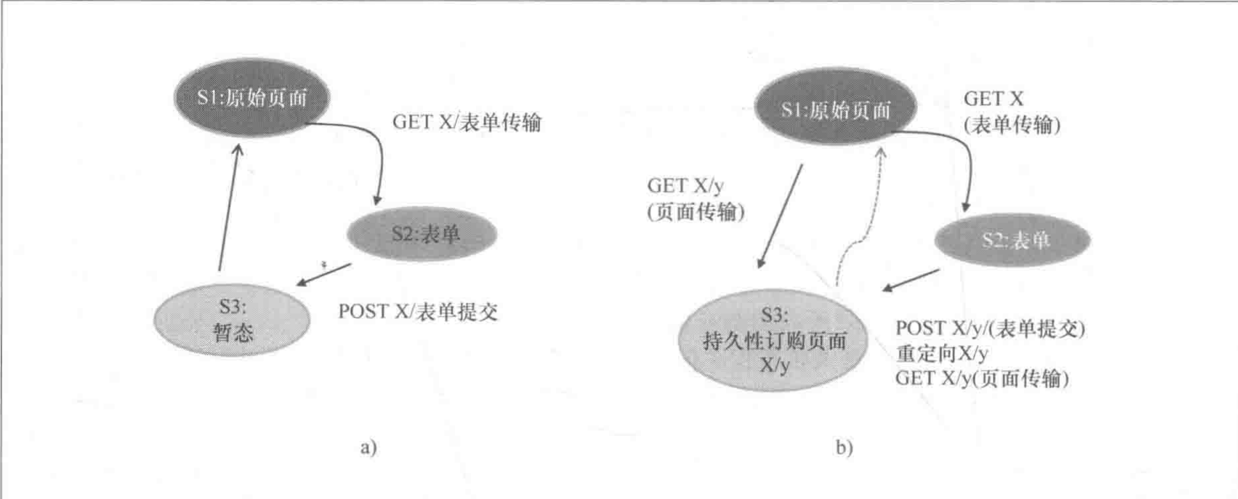


图 A. 14　消除暂态状态 a）具有暂态状态的服务 b）具有持久状态的同一个服务

A. 4　身份与访问管理机制

本节将进一步介绍第 7 章引入的身份与访问管理机制方面的内容。这些机制中大多数是标准化的。先介绍密码管理，之后介绍 Kerberos（身份验证），它被广泛应用于企业中的认证。事实上，Kerberos 是一个具有良好定义架构和通信协议的完整系统。它通过设计支持相互认证和单点登录。介绍了 Kerberos 内容之后，将回到访问控制的内容上来。我们首先回顾一下实现访问控制矩阵的两种常用方法：按照列和行将信息存储在非空单元格中。列方法产生访问控制列表，而行方法产生能力列表。接下来，对 Bell – LaPadula 模型（高级访问控制技术的基础）进行回顾。之后，将对处理身份联盟的一些技术，例如安全声明标记语言（Security Assertion Markup Language，SAML）、OAuth 2. 0 和 OpenID Connect 进行讨论。最后，将论述支持基于策略访问控制的可扩展访问控制标记语言（Extensible Access Control Markup Language，XACML）。基于 SAML，XACML 旨在规定访问控制策略和查询方面的规范。



A.4.1 密码管理 ★★★

使用密码进行认证是有问题的，部分原因在于内存方面存在的限制。最好的做法要求密码的长度足够长（例如，长度超过10个字符），包含非字母数字字符，看起来似乎没有意义并且做到定期更改等。但是，这种被认为足够强大的密码是难以记住的。随着云服务的发展，这种情况只会逐渐恶化。不止一个密码，每个人都会有很多密码，无法记得它们。过去，忘记密码时，用户可以呼叫客户支持中心。现在，用户可以通过在预先注册的电子邮件地址中收到的Web链接重置密码[㊀]。鉴于端到端电子邮件的漏洞，密码重置步骤可能包括辅助认证的形式，但是这会不可避免地带来额外的副作用。通常，辅助认证是基于之前用户对一系列可选问题给出的答案实现的。"你母亲姓什么?"是一个标准的问题。一般来说，这些问题是与用户有关的一些常识。这就出现了一个困境，一方面，为了最大限度地提高用户记住答案的可能性，而不需要把它们写下来，所以只有真实的（或简单的）答案应当被提供给系统。另一方面，在Google时代这样的答案往往很容易被别人知道。这种困境可以通过让用户自己定制安全问题从而实现其答案的鲜为人知性来轻松解决。例如，如果用户在高中时有一位外号为"乌龟"的地理老师，那么用户可以设置问题为"乌龟教什么?"。此外，还可以增加安全问题的数量。总的来说，安全问题的答案基本上是另一个秘密。这个秘密是相对长期的，实际上也需要得到相应的保护。

在密码方面，还有一个主要问题与如何将它们存储在认证系统中有关。以明文形式保存进行正常的访问控制显然是不够的。进入身份认证系统的攻击者可以轻松窃取存储的密码并冒充用户。而且，对存储密码有合法访问权限的系统管理员也可能会有不当的行为。标准做法是将密码进行哈希加密，并仅存储哈希值。哈希加密是单向实现的，但这种表述不是正式定义的概念。笼统来说，如果一个函数$H(p)$是单向的，那么它必须具有能够从p"很容易地"计算出$H(p)$，且从等式$H(p)=Q$"非常困难地"解出p的属性。即使知道一种这样的解法，找到另一个解法仍然是"非常困难的"。当然，"容易"和"困难"的描述并不准确。它们所指的是计算复杂性。如果可以计算得很快（例如，在几秒钟之内），那么它是"容易的"，但是如果没有已知的算法可以用来计算或在现代计算机上需要耗费1000年以上的时间进行计算，那么可以认为它是"困难的"。

使用密码哈希方案，验证用户输入的密码，认证过程计算它的哈希值，然后将该结果与存储的密码哈希值进行对比[㊁]。因此，没有人，甚至根本不可能查到用户密码的明文，这是一个壮举。例如，窃取到加密的密码无法实现相同的效果，因为它是不可逆的。谁都可以访问到加密的密钥，无论是通过合法的方式还是非法的方式，但却无法知道它所对应的加密之前的密码。

Roger Needham 和 Mike Guy 被认定在1963年发明了密码哈希[19]，但标准加密哈希算法（例如，MD5[㊂]和SHA-256）出现得很晚。一段时间以来，人们只能使用"自制"的算法。一个这样的算法在Multics中实现，并在后来的代码审查期间被发现是有缺陷的，这也证明了加密技术是一种难度很大的领域，需要人们共同的智慧。Multics事件的发生促使Robert Morris和Ken Thompson为UNIX系统开发了一种新的哈希函数[20]，该函数最终在2000年被证明是加密安全的[21]。

尽管它有很多的优点，但密码哈希方案仍容易受到字典攻击。这种攻击使用预先构建的潜在密码（例如，来自牛津英语词典中的通用名称和单词）组成的字典，以及使用已知算法计算出的哈希值。字典需要时间来构建，但这只需要做一次。供攻击者设法获取哈希密码，然后查找字典中的哈希值。如果有匹配，那么密码即被识别。字典攻击的问题很严重，因为总有人选择易破解的密码，很容易通过字典查找出来。

幸运的是，在20世纪70年代贝尔实验室开发UNIX操作系统时，Robert Morris和Ken Thompson就已经预料到了这些攻击，并设计了一种对付它们的技术[20]。该技术依赖于计算密码的哈希值时其中包含的n位随机数（称为盐值）。换句话说，执行哈希计算的不仅仅是密码，而是密码与盐值的连接。盐

㊀ 有些网站支持密码恢复。为此，网站保留用户密码，管理员可以知道它们，所以应该远离这样的网站。

㊁ 两个不同的密码产生相同的哈希值是不可能的。否者，就有存在漏洞的可能。这里有一个典型的例子，Alice发现她密码的哈希值与Bob的相同（通过读取操作系统的密码文件），并可以使用自己的密码以Bob的身份进行登录。在这种情况下，一种补救的方法是将哈希密码的访问限制为只有特权用户才可以进行。这种方案在UNIX中被实现为影子密码。

㊂ 人们已经证明，MD5的防碰撞能力不足，因此不推荐使用。

值特定于每个密码，并且当密码更改时，盐值也随之更改。盐值以明文的形式与哈希值一起存储。当验证用户输入的密码时，认证过程将查找盐值，计算盐值与密码组成连接的哈希值，并将结果与存储的值进行比较。因此，纯密码哈希字典不再有效，必须针对每个盐值重新构建密码字典。使用 n 位的盐值，意味着会有 2^n 个新密码字典。当这种方法在 UNIX 操作系统中首次引入时，选取的盐值长度为 12 位。如今，使用更为强大的处理器和更加便宜的存储设备，盐值的长度应该被设置得更长（至少要达到 64 位的长度），从而增加预计算的成本。还有一种减轻字典攻击的方法是多次迭代哈希操作。然而，这种方法对运行时认证的性能有一定的影响。

A.4.2 Kerberos ★★★

最初，Kerberos 被设计为可在分布式环境中的任何工作站（相对于用户自己的机器）上基于用户所知内容验证用户的身份。其主要目的是提供在任何计算机的用户和属于该网络的任何指定资源（服务器）之间提供相互认证。鉴于 Kerberos 内置的单点登录支持功能，Kerberos 一直是企业界选择的解决方案。Kerberos 是在麻省理工学院开发的㊀，IETF 完成了它的标准化工作。RFC 4120 中规定了 Kerberos v5 的核心规范。今天的大多数操作系统都支持 Kerberos㊁，其中包括微软™的 Windows 操作系统。

基本上，Kerberos 中的用户身份认证是基于密码的。但是，密码不再直接交换。这是通过基于 Needham - Schroeder 协议[22]方案实现并完成的，该协议使用加密操作来实现相互认证和机密性保护。该方案的核心是密钥分配中心（Key Distribution Center，KDC），它与管理域（称为域）中的每个用户（以及每个服务器）共享密钥。图 A.15 展示了该方案的工作原理。

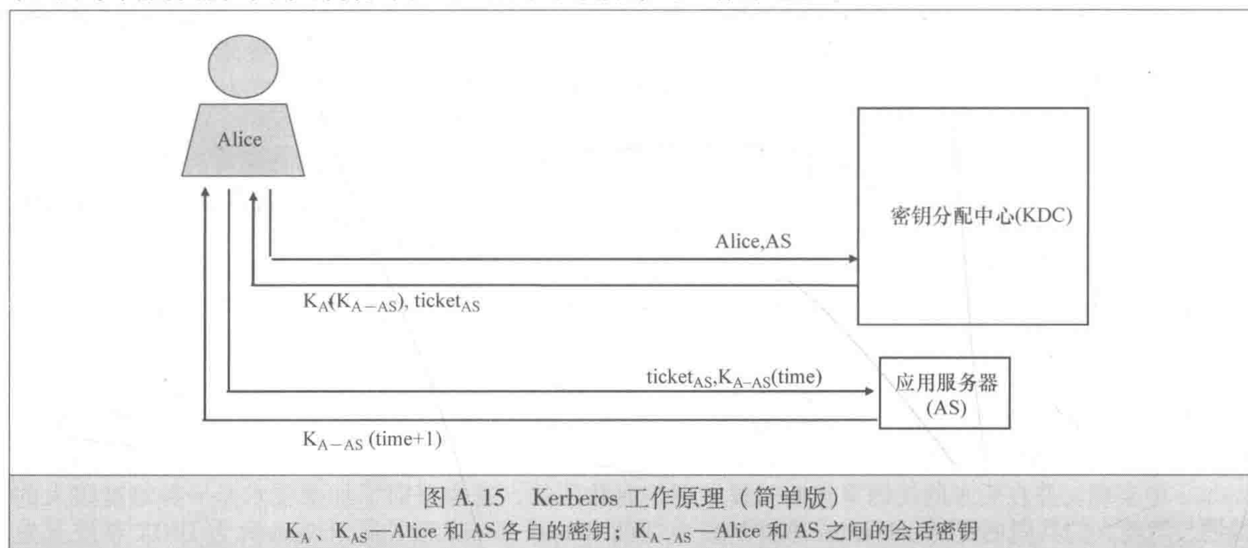


图 A.15 Kerberos 工作原理（简单版）
K_A、K_{AS}—Alice 和 AS 各自的密钥；K_{A-AS}—Alice 和 AS 之间的会话密钥

例如，Alice 想要访问提供电子邮件服务的应用程序服务器。她登录到 KDC，提供了她的名字和应用服务器的名字。在收到 Alice 的信息之后，KDC 为 Alice 和该应用服务器生成了一个会话密钥（K_{A-AS}），供两者共享，并使用 Alice 的密钥对该密钥进行加密［产生 K_A（K_{A-AS}）］，同时使用应用服务器的密钥也对该密钥（以及 Alice 的名字）进行加密（产生 $ticket_{AS}$，称为应用服务器票据），并将这两个加密的 blob（二进制大对象）发送给 Alice。Alice 解密会话密钥，然后使用该会话密钥对时间戳进行加密，并将其与票据（对 Alice 来说是不可读的）一起发送给应用服务器。通过解密票据，应用服务器得到了会话密钥和 Alice 的名字。现在 Alice 和该应用服务器都装有会话密钥，它们可以通过证明密钥的知识来相互认证。这是通过 Alice 发送加密的时间戳和应用服务器以同样的方法进行响应来实现的，除了增加时间戳用以避免重放攻击。Alice 成功完成相互认证后获得服务。只要票据有效，Alice 就

㊀ 麻省理工学院还开发了 Kerberos 开源软件，目前由它的 Kerberos 和互联网信任联盟（Kerberos and Internet Trust Consortium，KIT）管理。

㊁ 其中一些实现基于麻省理工学院开发的开源软件，该软件还托管了 Kerberos 和互联网信任联盟来对 Kerberos 软件项目进行管理。

可以使用相同的票据获得后续的服务。如果票据失效，那么她需要得到一张新的票据。

从用户界面的角度来看，密钥是透明的。Alice 在登录 KDC 时仍然需要提供她的姓名和密码。Kerberos 的工作是将密码转换成密钥，并立即将其删除（当然，每当 Alice 改变密码时，她的密钥就需要重新导出，并且相应地更新 KDC。更改密码实际上可以实现为通过 Kerberos 票据访问的服务）。但这种认证流程存在两个问题，如图 A. 15 所示。第一个问题是，KDC 无法知道 Alice 是否真地发送了请求。不管怎样，它发送了一个回复。这是无害的，因为除了 Alice 没有人能够理解这个回复。然而，攻击者只能继续发送攻击 Alice 密钥的请求，尤其在它是从密码导出的情况下。为了解决这个问题，当 Alice 发送请求时，她可以在她的密钥上包含一个加密的时间戳。现在，KDC 就可以知道是否该发送者是真的 Alice，并且只有在这种情况下，才会发送回复。

另一个问题是，客户端必须在整个登录会话期间记住 Alice 的密钥，以便在需要的时候证明她的身份。这使得这种长期密钥容易受到攻击。为了解决这一问题，需要在 KDC 中引入一个特殊的应用服务器作为缓冲区，被称为票据授予服务器（Ticket - Granting Server，TGS），负责向其他的应用服务器颁发票据，并与 KDC 中的认证服务器共享密钥，如图 A. 16 所示。现在，当 Alice 首次登录时，会得到一个短暂的会话密钥（即 K_{A-TGS}）和一个特殊的票据（称为票据授予票据）。每当 Alice 寻求访问服务器的权限时，客户端都将使用 K_{A-TGS} 和 $ticket_{TGS}$ 代表她与 TGS 进行交互。只要 K_{A-TGS} 和 $ticket_{TGS}$ 有效，就可以用它们得到一个票据，而无须 Alice 重新输入她的密码。

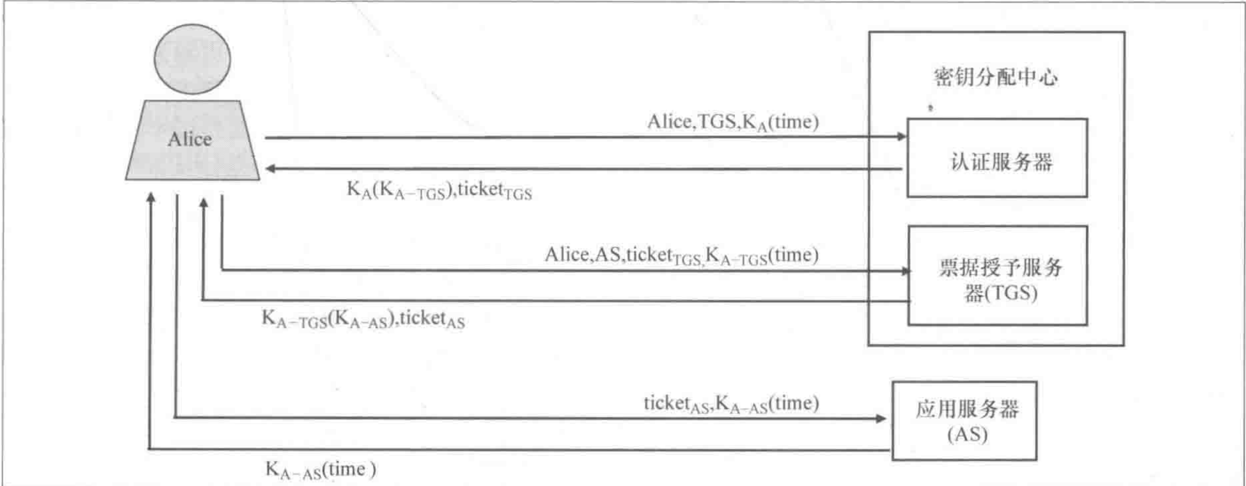


图 A. 16 Kerberos 工作原理（改进版）

K_A、K_{TGS}、K_{AS}—Alice、TGS 和 AS 各自的密钥；K_{A-TGS}—Alice 和 TGS 之间的会话密钥；K_{A-AS}—Alice 和 AS 之间的会话密钥

Kerberos 还支持向其他域提供票据（例如，企业外部的组织通过它与企业建立关系）。所产生的跨域认证对用户来说是透明的，用户在输入主机域的用户名和密码后，不需要重新认证身份。它工作的方式是将一个域（例如，域 B）TGS 注册到另一个域（例如，域 A）。然后，Alice（属于域 A）可以通过首次从域 A 中的 TGS 获得用于域 B 中 TGS 的票据，然后使用该票据从域 B 中的 TGS 获得用于应用服务器的票据，从而来访问域 B 中的应用服务器。

总而言之，Kerberos 的重要功能是：

1）基于他/她所知道的（即密码）对人类用户进行认证，用户可以随意更改这些他所知道的内容（例如，密码）。

2）支持从网络外的主机对所有网络资源的单点登录。

3）将用户与协议复杂性相分离，用户不需要了解和考虑其中的复杂过程（包括那些与密钥生成和管理相关联的协议，它们实际提供了比简单密码方案更为强大的认证功能）。

4）通过确保永久密钥永远不会存储在网络密钥分配中心之外的地方，来保护终端用户。

A. 4. 3 访问控制列表 ★★★

访问控制列表（ACL）是特定于对象的。访问控制列表指定了可以访问给定的对象的主体以及主体的权限。该列表通常作为操作系统的一部分进行集中保存和管理。当主体试图访问对象时，中央系

统将搜索与这个对象关联的 ACL。如果主体连通必要的权限在列表中，则可以进行访问。否则，访问被拒绝。图 A. 17 展示了与第 7 章中的访问控制矩阵示例相对应的 ACL。可以看到，ACL 可能仍然是冗长的。一方面，ACL 的大小随着合格用户的数量而增长，如果用户群体频繁变化，那么 ACL 就会受到影响。因此，如果该表中的冗余信息可以减少，将会起到一定的作用。一种方法是使用群组（Group）㊀的概念。群组由共享相同权限的多个主体组成。因此，ACL 只面向群组而不是群组中的每个主体。在这个意义上，群组是一个特殊的主体。常规主体可以被分配到一个或多个群组中，并且可以根据单个主体的权限或群组的权限来访问对象。

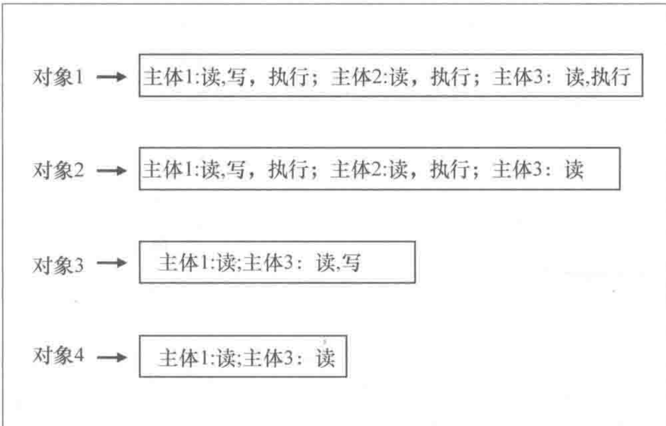


图 A. 17　访问控制列表

ACL 被 Multics 使用，并在 UNIX 操作系统中得到了广泛的应用㊁。与文件相关联的 ACL 通常表示为三个三元组，分别识别文件拥有者（Owner）、群组（Group）和其他用户（Other）的权限。每个三元组由分别控制文件是否可以被读取、写入和执行的标志组成。设置了所有标志的文件所拥有的 ACL 可以写成 rwxrwxrwx，表示该文件对所有用户（包括 Owner、Group 和 Other）可读、可写和可执行。具有更严格访问限制的文件的 ACL 可以写成 rwxr - - - - - -，表示该文件只对其拥有者（Owner）可读、可写和可执行，它限制了拥有者可以分配的权限。例如，Alice（作为拥有者）不能允许 Bob 读取她的文件，Chris 写入她的文件，Debbie 读写她的文件，以及 Eve 执行她的文件。所有基于 UNIX 的操作系统都已经添加了具有不同复杂程度的 ACL 功能。

总的来说，使用 ACL，在访问时可以很容易地验证特定用户是否确实有访问的权限。对象的拥有者也可以很容易地撤销给予对象的权限。拥有者只是从对象的 ACL 中删除主体的权限即可。对给予主体的权限进行降级也很容易。只需将主体在 ACL 中相应条目内的具体权限删除即可。但是，ACL 有局限性。起初，ACL 不适合处理用户需要将权限委派给其他用户一段时间的情况，例如，主管要求下属在度假期间批准采购请求。还存在从众多 ACL 中确定每个用户的权限的内在难题。然而，当某个用户访问权限需要被撤销时，这样的决定是必要的。ACL 可以同时包含用户和用户所属的群组。如果用户对对象的权限被撤销，那么该用户仍然可以通过群组成员身份访问这一对象。

A. 4. 4　能力列表



能力列表是特定于主体的，它包含了授予给主体的能力。能力指定了特定的对象以及对该对象允许的操作。Dennis 和 Van Horn[23] 在描述一种控制内存中对象访问机制时，引入了这一术语。从概念上讲，能力类似于前面介绍的 Kerberos 票据或 OAuth 令牌。图 A. 18 展示了与第 7 章中的访问控制矩阵示例相对应的能力列表。主体的能力集明确决定了主体可以访问的对象，而不需要对主体进行认证。

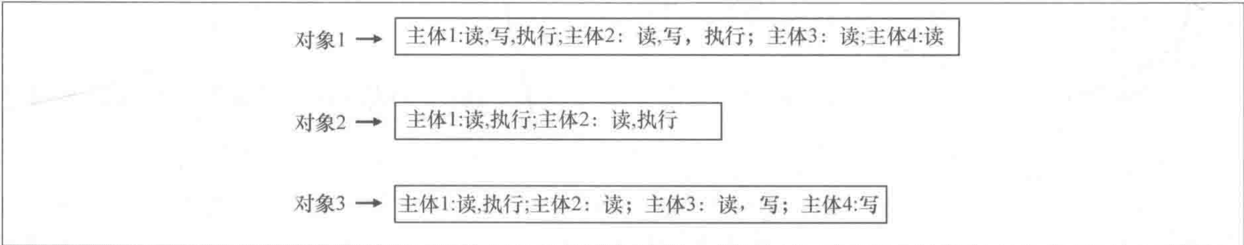


图 A. 18　能力列表

㊀ 这里定义的群组类似于角色，其中与角色定义的细微差异在于群组可以让它的成员基于任意的规则。因此，群组的含义要比角色的含义更为广泛，角色往往被绑定到一组特定的权限上（例如，作业功能）。

㊁ ACL 自然是操作系统的一部分。要让 ACL 工作，命名和认证主体（用户、进程等）是必须的，并且这两者都已经被操作系统处理。

当对一个预期的对象进行访问时，对象必须呈现出一种相应的能力。通常，主体会提前获得该能力，并将其存储供以后使用㊀。因此，主体必须不能伪造或修改能力，然后再使用它。换句话说，能力应该是防篡改和可认证的。为此，有效的方法是通过加密技术来实现这一目的。相关的例子是 OpenStack Keystone 中的 PKI 令牌，它们被签名并且是可验证的。

还有一个例子，Andrew Tanenbaum 等人[24]为 Amoeba 分布式操作系统开发了一个方案。该方案工作原理如下，除了通常的对象标识符和权限之外，能力中还包含一种可认证的校验和。该校验和使用一种加密安全的单向函数（如 HMAC）在对象标识符、权限和仅对访问控制系统已知的密钥（实际上是一个随机数）上进行计算得出。当试图对对象进行访问时，主体将该能力作为请求的一部分发送到系统。系统使用能力中的对象标识符和权限以及它所持有的密钥来计算校验和，如果校验和与能力中的校验和匹配，则该请求被授予。否者，被拒绝。如果主体更改了对象标识符或权限，则能力中的校验和将变为无效。没有密钥，主体也无法产生正确的校验和。这种方案应当会让读者想到第 6 章讨论的对象存储访问控制机制，其工作原理与此基本相同。

基于能力系统的另一层含义是，主体可以将能力的副本传递给其他的主体，而无须交换授权，从而使它们能够访问对象。这是一把双刃剑。一方面，共享和委派变得更简单了。另一方面，跟踪谁将哪些能力给了谁，以及验证被给予能力的新持有者是否经过授权则变得更加困难。因此，难以将对象的选择性主体的访问权限撤销。一种解决的方法是让与该对象关联的所有尚未解决的能力失效，并向合格的主体授予新的能力。这可以通过改变 Amoeba 中的密钥和保持对 OpenStack Keystone 中的撤销事件的跟踪来实现。

A.4.5 Bell - LaPadula 模型 ★★★

Bell - LaPadula 模型[25]通过两种策略规则（称为属性）来处理对信息流的控制。一种策略规则是简单安全属性，其中主体只能读取相同或较低安全级别的对象。因此，将军可以阅读士兵的文件，而士兵不能阅读将军的文件。但是这种不可上读（no - read - up）规则不足以阻止信息向下级泄露。将军可以阅读机密文件，并将他阅读的内容写到士兵可以阅读的未加密文件中。为了防止这种情况发生，不能允许主体向下面的安全层次写入读取的内容。因此，限制属性或* - 属性㊁产生了，这种属性假定主体职能在相同或更高安全级别上修改对象（这项规则可以防止将军复制机密文件内容并将它粘贴到未加密文件中的情况发生）。

当执行这两个规则时，信息只能向上流动，如图 A.19 所示。但这在实践中是有问题的。在某些时候，部队必须从他们的指挥官那里了解他们要去哪里。Bell 和 LaPadula 通过从这些规则中豁免一组特殊的受信任的主体解决了这个问题。另一个 Bell - LaPadula 模型的假设是，涉及参与者的安全级别保持相同（这被称为基本假定性质）。如果安全级别允许改变，那么信息可能会流向不需要的方向。图 A.20 展示了这种情况，其中主体 4 在从上一个示例中的对象 3 读取后，降低了其自身的安全级别，并写入对象 1，从而导致敏感信息被泄露。实际上，基本假定性质有两个版本：强的和弱的。强版本禁止在系统操作期间更改安全级别，而弱版本则允许更改，只要不违反建立的安全策略（例如，"不可上读"或"不可下写"）。

针对保密，Bell - LaPadula 模型本身不关心秘密的可信赖性和完整性，这里存在一个风险。基于这种模型的应用，士兵可以重写他上级的情报报告。因此，即使依靠只有将军才可以阅读报告的方法，将军仍然有可能得到错误的信息。这个问题的关键在于主体创建的对象的可信度可以降级，但不会被主体读取的对象的可信度升级。Ken Biba 通过将 Bell - LaPadula 模型颠倒过来解决了这个问题。他的模型[26]引入了两个相反的属性：上读和下写。与 Bell - LaPadula 的属性类似，它们的命名如下：

1）简单完整性属性，指的是主体只能读取相同或更高安全（或完整性）级别的对象。

2）* - 完整性属性，指的是主体只能写入相同或更低安全（或完整性）级别的对象。

这些属性体现了"低水位原理"：由其他对象组成的对象的完整性与这组对象中最不可靠的对象的完整性相同。从本质上来说，这些属性反映了策略规则。执行时，确保维护信息的完整性。Biba 模型

㊀ 这与基于 ACL 的系统完全不同，该系统本身需要管理列表。

㊁ 作者使用"*"作为临时占位符，等待有令人满意的名字出现时将其代替。

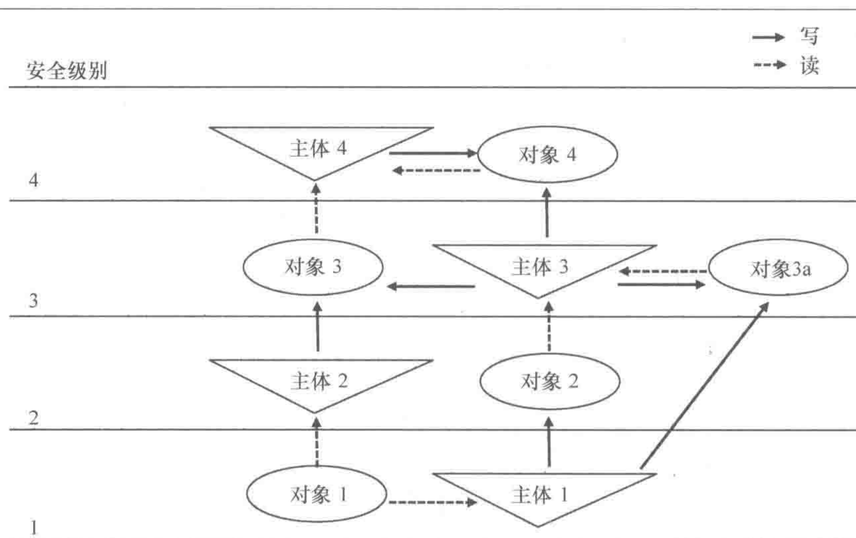


图 A. 19　Bell－LaPadula 系统中的信息流

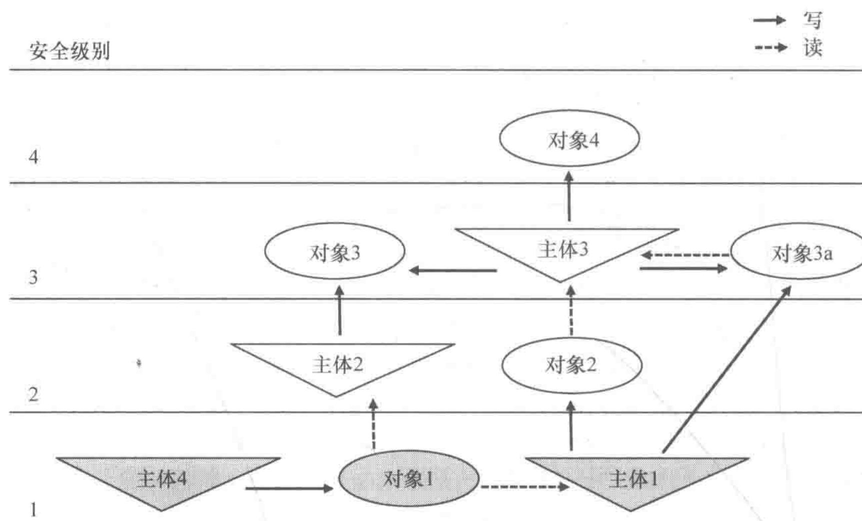


图 A. 20　Bell－LaPadula 系统中更改的信息流

是第一个基于信息流完整性的正式的、可验证的模型。与 Bell－LaPadula 模型一样，信息的严格单向流动使得直接将 Biba 模型应用到实际应用中变得困难。通常情况下，需要打破这种信息流限制，并且必须根据具体情况进行操作[㊀]。此外，信息披露和完整性的保护通常需要一起处理，但是 Biba 和 Bell－LaPadula 模型是相矛盾的，从而阻止了安全级别之间的通信。

A. 4. 6　SAML ★★★

SAML 是基于 XML 的一种广泛实现的标准。最初它是由 OASIS 开发的，之后通过一系列的努力被人们所采纳。特别是自由软件联盟的身份认证联盟框架（Identity Federation Framework，ID－FF）项目[㊁]和 Shibboleth 开源项目同时采用了 SAML 1.0 版本，而 OASIS 此时还正在对这一标准进行修订。由提供

㊀ 实际上，微软的 Windows Vista 系统实现了一种默认的“不可上写”的策略。将 IE 浏览器设置为低安全性级别。但是，默认情况下，对象具有中等安全性级别。这是为了限制基于浏览器的攻击造成的损害；IE 无法打开任何用于写入访问的目录、文件或注册表项，除非它们被明确设置为低安全性。

㊁ 自由联盟已经被整合为坎塔拉倡议，这是一个“促进身份社区协调统一、互操作性、创新和广泛采用的”论坛。

商需求驱动的 ID - FF 项目引入了信任圈子的概念，并建立在 SAML，以实现其中的身份联合。相比之下，Shibboleth 项目主要处理单点登录和隐私保护访问控制方面的问题，因为它来源于面向教育和研究的互联网。然而，人们对 SAML 所做多方面的努力，却导致了不兼容变体的产生。幸运的是，利益相关者能够及时地组织起来，纠正这个过程并制定了一种统一的版本，其结果是 SAML 2.0 由 OASIS 在 2005 年批准。这是大多数 SAML 支持的版本。

SAML 2.0 由一系列规范组成，其核心规范定义了安全声明概念和在身份识别提供者与信赖凭证方之间进行声明交换的协议［该核心规范作为 ITU - T Recommendation（建议）X.1141[28] 于 2006 年发布］。安全声明通常由身份识别提供者发布，并由信赖凭证方使用，从而对主体（例如，终端用户）进行认证和授权。声明由一组关于主体和上下文信息的语句组成，例如，发行者、接收者、发行时间和到期时间。语句可能涉及认证、属性或授权决策。认证语句描述了认证交易，包括相关信息，例如认证方法和交易时间（SAML 2.0 没有规定特定的认证方法，并且可以支持一系列具有不同优势的认证方法，例如密码、Kerberos 票据和 X.509 证书）。属性语句描述了与主体相关联的属性。最后，授权决策语句声明了是否允许主体访问请求的资源。为了确保完整性，SAML 声明由发行者进行数字签名。

SAML 2.0 定义了一组请求响应协议，每个请求响应协议都是为特定目的定义的。例如，声明查询和请求协议可以让信赖凭证方查询或请求关于主体的 SAML 声明（相关的属性、认证或授权决策）。认证请求协议可以让主体从身份识别提供者获得认证声明。命名身份映射协议可以让信赖凭证方从身份识别提供者获得新的主体标识符。然而，这些协议没有被定义在可以直接用于有关各方之间的通信层面上。它们必须依靠现有的通信协议（如 HTTP）来通过协议映射传送消息（即请求和响应）。在 SAML 命名中，将 SAML 消息映射到标准通信协议上被称为绑定[29]。例如，HTTP 重定向绑定定义了如何将 SAML 消息作为 URL 参数的一部分携带。由于 URL 的长度在实际中是有限的，因此需要专门的编码。而且，非常大的消息将需要通过其他的绑定来传输。一个这样的绑定是 HTTP POST，它可以让 SAML 消息作为 base64 编码内容的一部分进行传输。不用说，不管哪种绑定，传输都必须要得到保护（通常通过 TLS）。

图 A.21 展示了一个通过“SAML HTTP 重定向绑定”来完成身份联盟消息流工作过程的示例。这里假设了，信赖凭证方和身份识别提供者之间有一个预先建立的关系。该流的工作过程如下：当检测到用户（通过用户代理）请求资源访问尚未被认证时，信赖凭证方会发出一个 SAML 认证请求，这个 SAML 认证请求被重定向到身份识别提供者。然后，由身份识别提供者执行用户认证的步骤。同样，这些步骤是特定于所选的身份认证方法的，这部分内容超出了 SAML 的范围，在这里不做介绍。认证步骤完成后，身份识别提供者发送携带声明的 SAML 认证响应，该响应被重定向到信赖凭证方。根据收到的认证响应，信赖凭证方随即做出相应的响应。

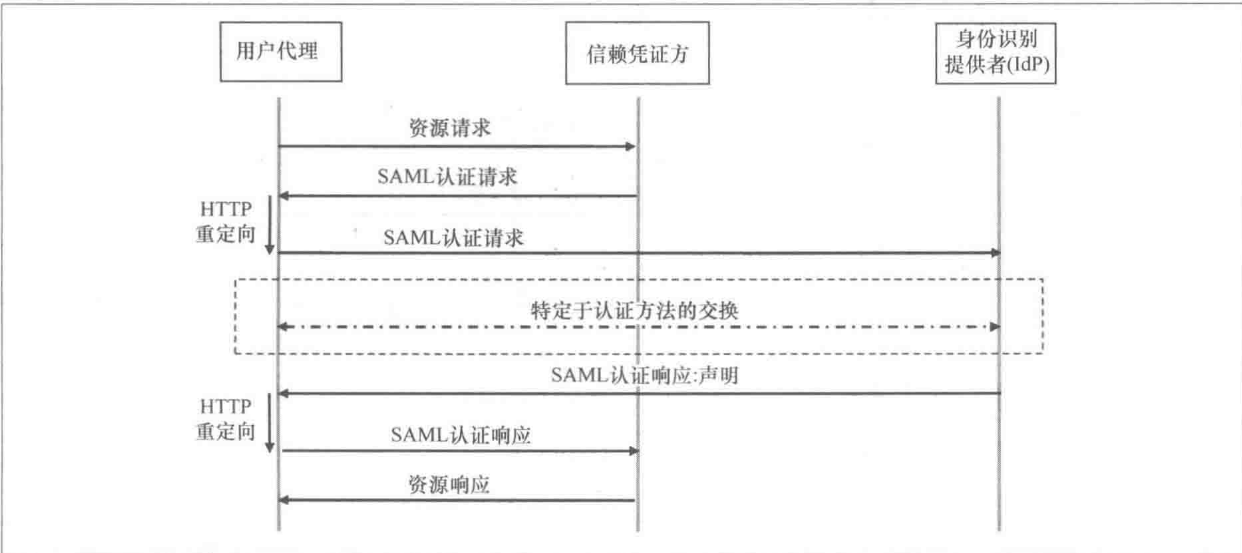


图 A.21　SAML 身份联盟消息流

A.4.7 OAuth 2.0 ★★★

本节将进一步介绍第7章提到的OAuth 2.0的细节内容。首先从OAuth 2.0定义的授权类型开始：

1）授权码授权。这种授权类型由授权服务器生成和验证，提供给客户端作为一次性使用，适用于客户端通过用户代理与用户交互的情况。授权码只能使用一次，并且绑定到客户端。当客户端获取访问令牌时，这种与客户端的绑定需要授权服务器的认证。需要注意的是，授权服务器单独规定了授权码的结构，它负责颁发和验证授权码。

2）隐式授权。这种授权类型旨在优化浏览器客户端（以JavaScript实现）的性能。对于这样的客户端，没有确定的认证方式。因此，客户端可以直接获得访问令牌，从而节省了访问令牌所需的额外的步骤和授权交换。

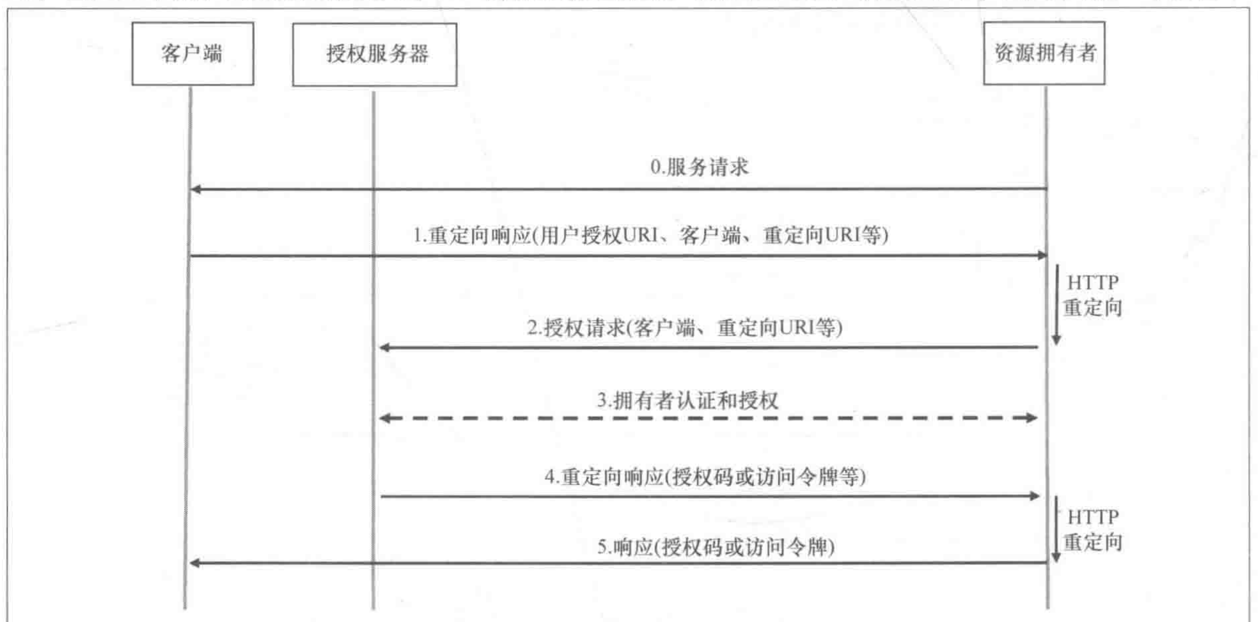
3）密码凭证授权。当然，这种授权类型似乎违背了OAuth的目标，即绝对不会泄露用户的密码。因此，只有当资源拥有者（提供密码一方）与客户端之间存在高度信任时（例如，客户端是用户设备的一部分），才应当使用这种类型的授权。

4）客户端凭证授权。这种类型的授权允许客户端通过事先安排来访问其控制的或授权服务器控制的受保护资源。

OAuth 2.0还允许人们定义新的授权类型。一种新兴的授权类型是SAML 2.0 Bearer Assertion（持有者声明）。该声明由身份识别提供者颁发（如前所述），声明包含了与主体有关的安全相关信息（例如，身份和权限），授权服务器可以使用这种信息。Bearer声明是一种特定类型的声明，使用它的持有者不需要提供任何其他证明（例如，加密密钥）。因此，这种声明必须得到适当的保护。考虑到不同参与者参与了声明的颁发与处理，需要一种标准的方法来对它们做出规定，即这里论述的SAML 2.0。

授权服务器通常需要客户端在颁发访问令牌之前对其自身进行认证。绑定到HTTP，OAuth 2.0支持基于密码（例如，使用HTTP基本身份认证或摘要认证方案）和声明的认证。显然，令牌请求和响应需要得到适当的保护。它们包含有敏感的信息（例如，密码、授权码和访问令牌）。用来实现这一目的的机制包括TLS 1.2、HTTP“缓存控制”首部字段（以防止敏感信息被缓存到HTTP缓存中）和将OAuth相关信息放在消息体而不是请求URI中发送（以防止敏感信息被记录在用户代理或其他可能的中间件上）。

为了优化用户体验，OAuth 2.0协议仅仅使用了HTTP重定向结构。图A.22进一步展示了相关的消息流。然而，使用HTTP重定向的代价较高。例如，假设用户可以使用所有可用于图形用户界面的技巧，但是对于用户来说验证重定向URI仍然是很困难的（如果不是不可能的话）。因此，用户可能在不



图A.22 OAuth 2.0用户授权消息流

知情的情况下授权非法网站。为了解决这个问题，一种对策是让授权服务器保存一份重定向 URI 白名单。换句话说，所有合法客户端均需要事先向授权服务器注册它们的重定向 URI。不过，这不是常用的做法。另一个问题是跨站请求伪造。从用户返回到客户端的重定向为攻击者提供了一种可注入攻击者自己授权码或访问令牌的渠道。一种对策是让客户端跟踪授权状态。为此，OAuth 2.0 支持一种在协议中携带状态信息的参数。当将用户代理重定向到授权服务器时，客户端可能包含此参数。然后，在所有相关的后续消息中重复，直到携带授权结果的消息回到客户端。由于参数的值由客户端单独设置和处理，所以客户端对其结构具有完全的控制权。在状态管理方面，它可以捕获授权会话标识符或某些特定的本地状态信息。在额外的安全保护方面，它可以包括签名。再次，由于敏感信息（例如，授权码和令牌）是通过网络传输的，所以授权交换应该通过诸如 TLS 的安全传输协议进行。

A.4.8 OIDC ★★★

OpenID 连接（OpenID Connect，OIDC）是 OpenID 的最新发展结果，两者之间唯一的联系只能体现在名字上。OIDC 是由 OpenID 基金会开发的，OpenID 是一个以用户为中心的身份联盟机制。使用 OpenID，用户可以选择和维护他的标识符（通常是 URI），并且可以选择他的身份提供者。OpenID 还配有自己的定制联盟认证协议，与 HTTP 紧密耦合[30]。与 SAML 相比，OpenID 拥有更加轻量级的方法，并且关注的范围更加集中。OpenID 仍然使用 XML 和自定义的消息签名方案，这使得它们在实现的正确性和互操作方面具有一定的挑战性。之后，社交网络服务就出现了，其相关联的身份联盟技术逐渐超越了 OpenID，并最终迫使 OpenID 继续前进。

OIDC 基于 OAuth 2.0，因此自然继承了后者的优点，包括支持 REST、JSON、标准签名和加密机制，以及各种部署情况。因此，在 OIDC 中，其互操作性得到了改善。但是以用户为中心的功能（例如，用户定义的标识符）消失了。根据 OpenID 基金会（该组织持续跟进 OpenID 的演进），OIDC：

"允许客户端基于授权服务器执行的认证来验证终端用户的身份，以及以客户操作的和类似 REST 的方式获得有关终端用户的基本配置文件信息。OIDC 允许所有类型的客户端（包括 Web 客户端、移动客户端和 JavaScript 客户端）请求和接收与认证会话和终端用户有关的信息。这套规范是可扩展的，允许参与者使用可选的功能，例如身份数据加密、OpenID 提供商发现和会话管理。"

图 A.23 展示了一个 OIDC 消息流的示例，其中基本遵循了 OAuth 2.0 中的授权码流。根据 OIDC 1.0 核心规范的内容，其中的主要差异包括：

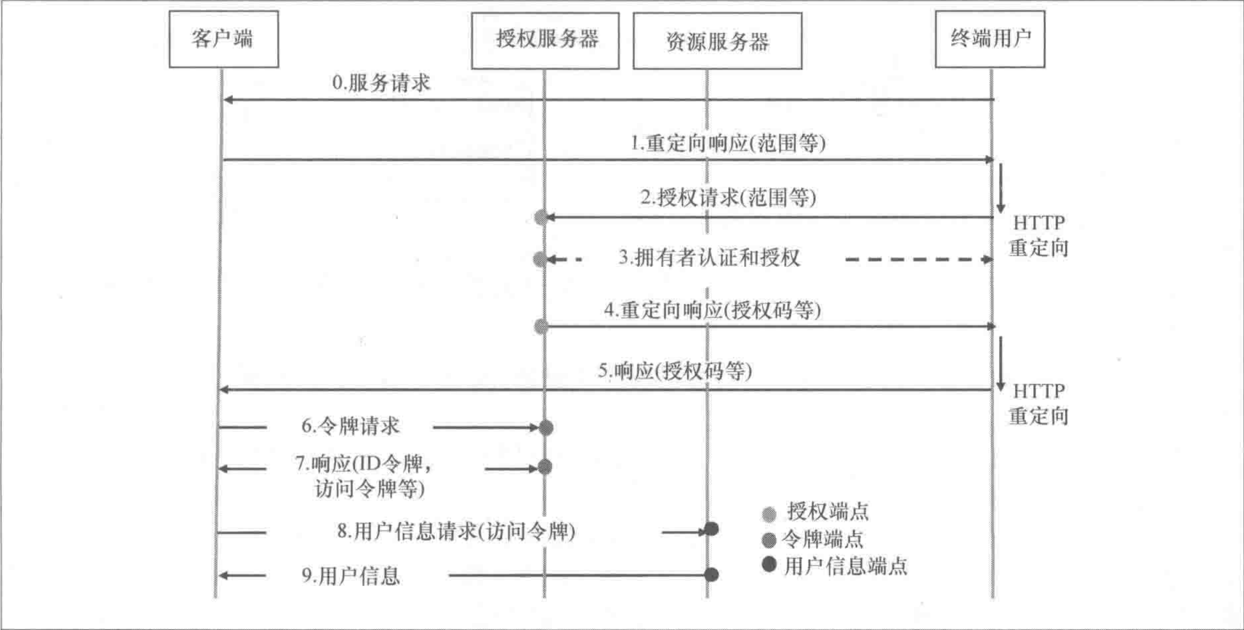


图 A.23 OpenID 连接消息流

1）为范围参数定义了一组特殊值（例如，开放的、配置文件和电子邮件）。在授权请求（例如，步骤 1 和步骤 2）中包含 openid 属于强制要求。还可以包含多个附加值。关于客户端可以获得的终端用

户的信息取决于这些范围值的存在。

2）除了访问令牌之外，ID 令牌也作为令牌响应的一部分予以返回（即步骤 7）。ID 令牌被表示成带有基于 IETF 标准的 JSON Web 签名的 JSON Web 令牌。它包含了一组由授权服务器做出的声明。这组声明必须包含验证响应发行者的信息、目标受众（即客户端）、令牌的到期时间和令牌的颁发时间。该信息用于在其签名被验证为有效之后进一步验证该令牌。

3）与经过身份验证的终端用户有关的声明被视为通过资源服务器的 UserInfo（用户信息）端点可访问的受保护资源。为了获得与终端用户有关的声明，客户端向 UserInfo 端点发送请求，其中包括访问令牌（如步骤 8 所示）。返回的声明取决于访问令牌中的范围（scope）值。例如，如果范围（scope）参数的值是配置文件（profile），则返回终端用户的默认声明。这些声明揭示了一些像全名、性别、出生日期和主页方面的信息。声明一般被表示成 JSON 对象，这些对象可以是签名的或加密的，或者既被签名又被加密。

A. 4. 9 访问控制标记语言 ★★★

基于属性的访问控制（Attribute - Based Access Control，ABAC）是从第 7 章讨论的 RBAC 模型演化而来的。它允许在访问对象时考虑不同的属性（在值和关系方面）。属性可以由对象提供，作为访问请求的一部分或从环境推断（例如，在时间和位置的情况下）。ABAC 的具体实现，即由 OASIS 和ITU - T 联合标准化的可扩展访问控制标记语言（Extensible Access Control Markup Language，XACML）[31,32]，该语言旨在规定访问控制策略和查询相关方面的功能。

XACML 遵循一般的策略控制模型，采用了前面描述的用于 QoS 支持的 PDP 和 PEP，并且还包括以下结构：

1）策略管理点（Policy Administration Point，PAP），负责管理调用典型操作（例如，创建、更新和删除）的策略。

2）策略库，它是一个存储策略的数据库或数据库集合（通常以规则的形式表示，如 IF <条件> THEN <动作>）。

图 A. 24 展示了这些不同的结构如何通过示例中的工作流程相互关联，其中包括以下步骤：

1）PEP 接收对受保护资源（或对象）的访问请求。

2）PEP 将请求传递给 PDP。

3）PDP 从策略库中获取使用的策略。

4）PDP 在做出访问决定之后，将结果返回给 PEP。

5）PEP 返回请求的资源或拒绝访问请求，执行决定。

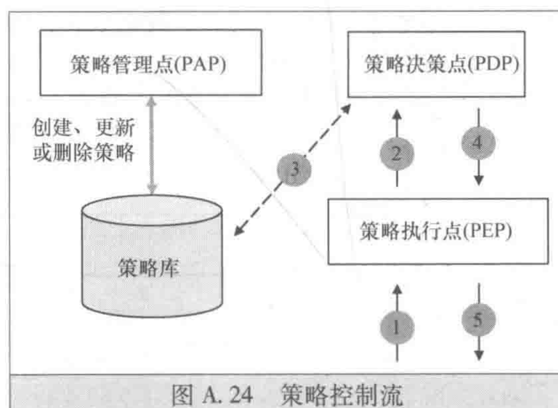


图 A. 24 策略控制流

XACML 基于 SAML 并与 SAML 一致。它包含两个关键组成部分：

1）基于 XML 的语言，用于表达授权和授权策略（例如，谁可以做什么，在哪里做，何时做）。此类策略存储在策略库中。

2）PDP 和 PEP 之间的请求和响应消息，其中请求消息用于触发并反馈到 PDP 中的策略评估过程，来自 PDP 的响应消息用于捕获 PEP 需要履行的动作或义务。

对于基于 XML 的语言，XACML 是冗长的，通常由机器生成。为了支持 RBAC，人们为 XACML 2.0 和 3.0 分别开发了两个同名的配置文件[33, 34]。在这两个配置文件中，角色一般被表示为主体属性㊀。根据应用程序环境，可能会有一个角色属性的值对应于不同的角色（例如，“雇员”“主管”或“官员”），

㊀ 具体来说，主体属性是与主体相关联的 XACML 请求中的 <Attribute> 元素。XACML 请求中的 <Attribute> 元素也可能与受保护的资源（资源属性）、对资源的动作（动作属性）或请求的环境（环境属性）相关联。



或不同的属性标识符，每一个都指示一个不同的角色。而且，两个配置文件中还定义了以下策略类型：

1）角色（Role），将一个给定的角色属性和值与一个权限相关联。

2）权限（Permission），包含实际的权限（即策略元素和规则）。

3）有特权的角色（HasPrivilegesOfRole），支持查询某个主体是否有与给定角色相关联的权限，也可以表达用户同时拥有多个角色的策略。

需要注意的是，在 XACML 2.0 的 RBAC 配置文件中，有一个额外的策略类型［即角色分配（Role Assignment）］被定义用来处理角色到主体的实际分配。但是，主体一般可以充当什么角色的问题超出了 XACML 的范围。该问题由角色启用机构根据下面的这段内容解决，即本章参考文献［33，34］中共同的范围描述内容：

“这样的实体可以使用 XACML 策略，但需要额外的信息……在该配置文件中指定的策略，假设提供给特定主体的所有角色都已经在请求授权决策时启用了。有一种情况是它们不处理的，即角色必须根据主体执行的资源或动作动态启用。因此，此配置文件中规定的策略也不会处理静态或动态的“职责分离关系”……未来的配置文件可能会满足这种情况的要求。”

参考文献

[1] Birman, K. P. (2012) Guide to Reliable Distributed Systems: Building High - Assurance Applications and Cloud - Hosted Services. Springer - Verlag, London.

[2] OASIS (2013) Committee Specification 01: Topology and Orchestration Specification for Cloud Applications, version 1.0. http://docs. oasis - open. org/tosca/TOSCA/v1. 0/cs01/TOSCA - v1. 0 - cs01. pdf.

[3] Prywes, N. S. (1977) Automatic program generation. Proceedings of National Computer Conference AFIPS '77, ACM, New York, pp. 679 - 689.

[4] Ahrens, J. and Prywes, N. (1995) Transition to a legacy - and reuse - based software life cycle. IEEE Computer, 28 (10), 27 - 36.

[5] Binz, T., Breiter, G., Leymann, F., and Spatzier, T. (2012) Portable Cloud services using TOSCA. IEEE Internet Computing, 16 (03), 80 - 85.

[6] Sunyaev, A. and Schneider, S. (2013) Cloud services certification. Communications of the ACM, 56 (2), 33 - 36.

[7] Waixenegger, T., Wieland, M., Binz, T., et al. (2013) Policy4TOSCA: Apolicy - aware Cloud service provisioning approach to enable secure Cloud computing. Lecture Notes in Computer Science, 8185, 360 - 376.

[8] Liu, K. (2013) Development of TOSCA Service Templates for provisioning portable IT Services. Diploma Thesis No. 3428, University of Stuttgart, Faculty of Computer Science, Electrical Engineering and Information Technology.

[9] Fielding, R. T. and Taylor, R. N. (2000) Principled design of the modern Web architecture. Proceedings of the 22nd International Conference on Software Engineering, ACM, New York, pp. 407 - 416.

[10] Fielding, R. T. (2000) Architectural styles and the design of network - based software architectures. PhD dissertation, University of California, Irvine, CA. www. ics. uci. edu/ ~ fielding/pubs/dissertation.

[11] Bush, V. (1945) As we may think. The Atlantic Monthly, 176 (1), 101 - 108. www. theatlantic. com/magazine/archive/1945/07/as - we - may - think/303881/.

[12] Nelson, T. H. (1965) Complex information processing: A file structure for the complex, the changing and the indeterminate. Proceedings of the ACM 20th National Conference, ACM, New York, pp. 84 - 100.

[13] Nabokov, V. (1963) Pale Fire. Lancer Books, New York.

[14] Rowberry, S. (2011) Pale Fire as a hypertextual network. Proceedings of the 22nd ACM Hypertext Conference, HT' 11, ACM, New York, pp. 319 - 324.

[15] Gray, H. J. and Prywes, N. S. (1959) Outline for a multi - list organized system. Proceedings of ACM '59; Preprints of Papers Presented at the 14th National Meeting of the Association for Computing Machinery, ACM, New York, pp. 1 - 7.

[16] Prywes, N. S. and Gray, H. J. (1963) The organization of a multilist - type associative memory. Transactions of the American Institute of Electrical Engineers, Part I: Communication and Electronics, 82 (4), 488 - 492.

[17] Barnet, B. (2013) Memory Machines: The evolution of hypertext. Anthem Press, London.

[18] Carmody, S., Gross, W., Nelson, T. H., et al. (1969) A hypertext editing system for the /360. Center for Computer & Information Sciences, Brown University, Providence, RI. File Number HES360 - 0, Form AVD - 6903 - 0,

pp. 26 - 27 (cited from [17]).

[19] Bonneau, J. (2012) Guessing human - chosen secrets. PhD dissertation, University of Cambridge.

[20] Morris, R. and Thompson, K. (1979) Password security: A case history. Communications of the ACM, 22 (11), 594 - 597.

[21] Wagner, D. and Goldberg, I. (2000) Proofs of security for the Unix password hashing algorithm. In Okamoto, T. (ed.), Advances in Cryptology—ASIACRYPT 2000. Springer, Berlin, pp. 560 - 572.

[22] Needham, R. M. and Schroeder, M. D. (1978) Using encryption for authentication in large networks of computers. Communications of the ACM, 21 (12), 993 - 999.

[23] Dennis, J. B. and Van Horn, E. C. (1966) Programming semantics for multiprogrammed computations. Communications of the ACM, 9 (3), 143 - 155.

[24] Tanenbaum, A. S., Van Renesse, R., Van Staveren, H., et al. (1990) Experiences with the Amoeba distributed operating system. Communications of the ACM, 33 (12), 46 - 63.

[25] La Padula, L. J. and Elliott Bell, D. (1973) Secure Computer Systems: Mathematical Foundations. MTR - 2547 - VOL - 1, Mitre Corporation, Bedford, MA.

[26] Biba, K. J. (1977) Integrity Considerations for Secure Computer Systems. MTR - 3153 - REV - 1, Mitre Corporation, Bedford, MA.

[27] OASIS (2005) Assertions and protocols for the OASIS Security Assertion Markup Language (SAML) V2. 0. http: //docs. oasis - open. org/security/saml/v2. 0/saml - core - 2. 0 - os. pdf.

[28] International Telecommunication Union (2006) ITU - T Recommendation X. 1141: Security Assertion Markup Language (SAML 2. 0). www. itu. int.

[29] OASIS (2005) Bindings for the OASIS Security Assertion Markup Language (SAML) V2. 0. http: //docs. oasisopen. org/security/saml/v2. 0/saml - bindings - 2. 0 - os. pdf.

[30] OpenID Foundation (2007) OpenID Authentication 2. 0. http: //openid. net/specs/openid - authentication - 2_ 0. html.

[31] International Telecommunication Union (2006) ITU - T Recommendation X. 1142: eXtensible Access Control Markup Language (XACML 2. 0). www. itu. int.

[32] International Telecommunication Union (2013) ITU - T Recommendation X. 1144: eXtensible Access Control Markup Language (XACML 3. 0). www. itu. int.

[33] OASIS (2005) Core and hierarchical Role Based Access Control (RBAC) profile of XACML v2. 0. http: //docs. oasis - open. org/xacml/2. 0/access_ control - xacml - 2. 0 - rbac - profile1 - spec - os. pdf.

[34] OASIS (2014) Core and hierarchical Role Based Access Control (RBAC) profile of XACML v3. 0. http: //docs. oasis - open. org/xacml/3. 0/rbac/v1. 0/csprd04/xacml - 3. 0 - rbac - v1. 0 - csprd04. pdf.

读者需求调查表

亲爱的读者朋友：

您好！为了提升我们图书出版工作的有效性，为您提供更好的图书产品和服务，我们进行此次关于读者需求的调研活动，恳请您在百忙之中予以协助，留下您宝贵的意见与建议！

个人信息

姓名：		出生年月：		学历：	
联系电话：		手机：		E－mail：	
工作单位：		职务：			
通讯地址：		邮编：			

1. 您感兴趣的科技类图书有哪些？

□自动化技术　□电工技术　□电力技术　□电子技术　□仪器仪表　□建筑电气

□其他（　　）以上各大类中您最关心的细分技术（如 PLC）是：（　　）

2. 您关注的图书类型有：

□技术手册　□产品手册　□基础入门　□产品应用　□产品设计　□维修维护

□技能培训　□技能技巧　□识图读图　□技术原理　□实操　□应用软件

□其他（　　）

3. 您最喜欢的图书叙述形式：

□问答型　□论述型　□实例型　□图文对照　□图表　□其他（　　）

4. 您最喜欢的图书开本：

□口袋本　□32 开　□B5　□16 开　□图册　□其他（　　）

5. 图书信息获得渠道：

□图书征订单　□图书目录　□书店查询　□书店广告　□网络书店　□专业网站

□专业杂志　□专业报纸　□专业会议　□朋友介绍　□其他（　　）

6. 购书途径：

□书店　□网站　□出版社　□单位集中采购　□其他（　　）

7. 您认为图书的合理价位是（元/册）：

手册（　　）　图册（　　）　技术应用（　　）　技能培训（　　）　基础入门（　　）　其他（　　）

8. 每年购书费用：

□100 元以下　□101～200 元　□201～300 元　□300 元以上

9. 您是否有本专业的写作计划？

□否　□是（具体情况：　　　）

非常感谢您对我们的支持，如果您还有什么问题欢迎和我们联系沟通！

地址：北京市西城区百万庄大街 22 号　机械工业出版社电工电子分社　邮编：100037

联系人：张俊红　联系电话：13520543780　传真：010－68326336

电子邮箱：buptzjh@163.com（可来信索取本表电子版）

读者需求调查表

姓名：		出生年月：		职称/职务：		专业：	
单位：				E－mail：			
通讯地址：						邮政编码：	
联系电话：		研究方向及教学科目：					
个人简历（毕业院校、专业、从事过的以及正在从事的项目、发表过的论文）							
您近期的写作计划有：							
您推荐的国外原版图书有：							
您认为目前市场上最缺乏的图书及类型有：							

地址：北京市西城区百万庄大街 22 号　机械工业出版社电工电子分社
邮编：100037　网址：www. cmpbook. com
联系人：张俊红　电话：13520543780　010－68326336（传真）
E－mail：buptzjh@ 163. com（可来信索取本表电子版）